Parallel Programming
Inofficial Course Script

Last Updated: April, 2023

Author: Sascha Kehrli

github.com/skehrli/ParallelProgrammingScript

ETH zürich

Table of Contents

1 Disclaimer

This script only covers the first lecture half (week 1-7). It is written to be read while following the lecture. Ideally, go to the lecture to get a broad overview from the professor and then read the chapter about the presented lecture topic in the script.

The script is written to very closely match the lecture contents from FS2023. It focuses on clarifying topics presented in the lecture in an intuitive way and, most importantly, streamlining and clearly connecting all the presented topics with frequent overviews of how certain things relate to previously learned concepts.

Currently, the script is not officially approved by the lecturers and they did not read or verify its contents. If you spot mistakes or notice that there are new contents from the lecture that are not covered in the script, do not hesitate to open a pull request on github (skehrli/ParallelProgrammingScript) or contact the author.

2 How concurrency is achieved: Processes and Threads

In this chapter, we want to explore the setting of the course. The final goal of the course is two-fold:
On the one hand, we want to be able to write multi-threaded programs in Java. But we also want to understand general concepts of concurrent programming and how to solve issues that arise with its introduction.
To get there, we first have to understand what a thread even is, how Java threads relate to them and how concurrency is achieved on the OS level.
Hence, before we start writing programs and analyze problems in concurrent programming, we take a look at how a modern OS handles programs and then define the setting the course takes place in. Note that this chapter explains multiprocessing in more depth than the course itself. It is not directly exam relevant, but the understanding gained in this chapter will be critical for later topics. At the end of the chapter, there is a short list of terms that are mentioned, but not explicitly explained in the text. When a term is not clear, it may be contained and explained in this list.

2.1 Threads and Processes from the OS Perspective

A modern computer can run hundreds of programs at the same time with only a few cores. The operating system sits between these programs and the hardware and is responsible for making this work. On a high level, the OS first needs some way of organising these programs to get them ready for execution. It can then run a program for a bit, stop it, run another for a bit and so forth, creating the illusion of many programs running at the same time. We look at how this is achieved in more detail and discuss some important concepts.

2.1.1 What is a process?

To start, we take a look at the lifecycle of a running program. Before the program is started, it is just a liveless collection of bits that sits on disk.
When we now start this program, the OS will make it come to live and view it as a *process*. To achieve this, the OS loads the code and static data of the program from disk to main memory.

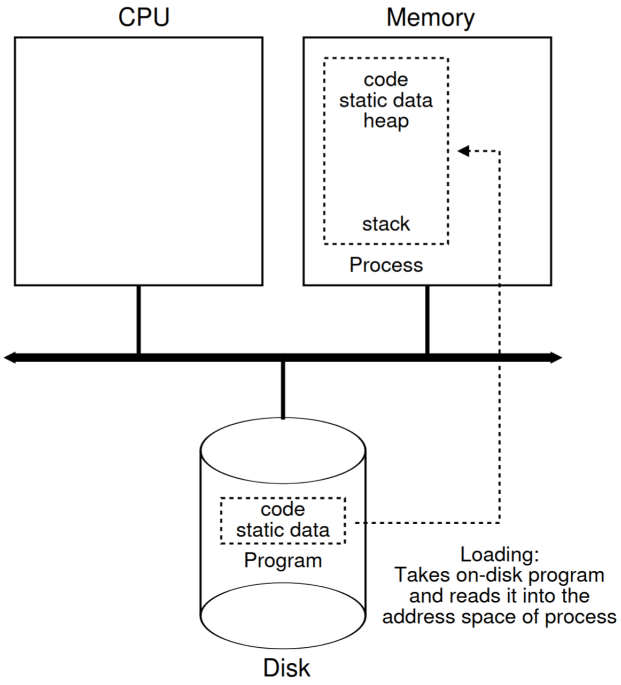


Fig. 1: Program to Process. https://web.archive.org/web/20210627072431/https://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/Book/cpu-intro.pdf

To actually execute its code, the process also needs memory to store variables and data structures. Hence, the OS initializes an address space for the process and allocates a stack and usually also a heap within this address space. Allocating here simply means that the OS reserves a chunk of memory and tells the process where it is by passing pointers. Initializing an address space means that each process will `only` see its own memory locations. It cannot access memory of other processes.

The OS also needs to do some other initialization related to I/O, such that the program can for example

read and write to a console or deal with files.

Going back to the beginning, we had a bunch of programs on disk we wanted to run at the same time. We transformed each of these programs into a process that has all the resources it requires to run loaded into main memory. The OS views each running program now as a *process* with an associated *context* or *state*, which includes all the things we mentioned before, like its address space, I/O information and CPU register state.

All we need to do now is choose one process and start executing its code on the CPU.

2.1.2 What is a Thread Now?

Most operating systems further divide processes into threads. Further dividing the work of a process makes sense, as large programs have many different tasks that need to be taken care of at the same time. If we think of a running Java program, we need for example a garbage collector running in the background.

In an OS that supports threads, each process consists of at least one thread and all threads within the same process share the same address space, that is, can view the same memory. This means that threads of the same process can easily communicate with each other, for example through shared variables. It also means that a thread has less *context* it needs to store than a process. This is because it inherits most of its state from the parent process, like the address space and I/O information. We can think of threads as smaller processes:

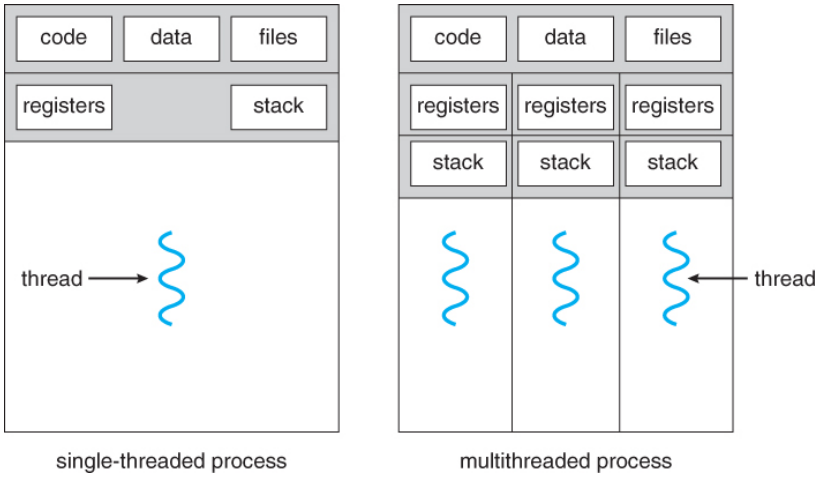


Fig. 2: Process vs. Thread. https://www.baeldung.com/cs/process-vs-thread

2.1.3 Scheduling and Context Switches

By introducing these OS concepts, the notion of parallelism boils down to how the OS divides the CPU cores on the competing processes and threads.
Imagine we have an n-core CPU. The OS can decide which process (and which of its threads) is running on each core at any time. The part of the OS responsible for this decision is referred to as the *CPU scheduler* or just *scheduler*. We can view the scheduler as a black box and just assume that it does a good job at giving sufficient CPU time to all processes and threads.
We assume a *pre-emptive scheduler*, which just means that the scheduler can interrupt a thread at any time and swap it out for another thread, even if the thread didn't finish executing all its instructions yet.

Descheduling a thread in favour of another is called a `context switch`. Upon a context switch, the OS has to store the context of the descheduled thread to memory and load the context of the newly scheduled thread from memory to the CPU core the thread is going to run on. Context switching threads is faster than context switching processes, but since a (thread) context switch is not meaningful work (as opposed to executing instructions of a running program), the OS still wants to minimize them.

Imagine we have a process with three threads and only a single CPU core. Only a single thread can be scheduled at a time, as illustrated here:

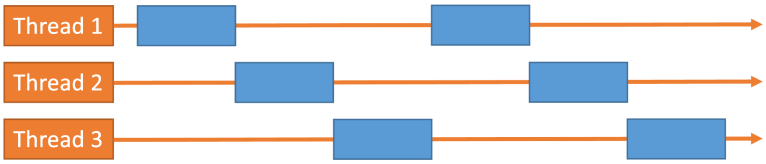


Fig. 3: Multiple threads context switching on one core.

When we now have three CPU cores, all of these can be scheduled in parallel. For this to happen, the scheduler first has to schedule the process of the Java program on all three cores (remember, processes *and* threads are scheduled). Then, all threads can run in parallel on separate cores:

Fig. 4: Each thread runs on a separate core.

To context switch, we need to store everything a thread requires to continue executing its code later on. We now know how such a context switch can work: Since a thread inherits the address space from its parent process, it only needs to remember the CPU register contents. Note that the CPU registers include special registers like the instruction pointer and the stack pointer. Imagine we now switch out the thread for another (within the same process) and then switch back. Simply restoring these registers is enough to continue its execution with the next instruction, since the thread has the variables both on its stack and in the registers back and knows which instruction to execute next (since stack pointer, instruction pointer and other registers are restored). Note that the exact set of steps needed for such a switch depends on the OS, but generally, this is what is required.

However, switching to a thread within another process is not as straightforward and we need to context switch to the other process first. Remember here that process context switching is a lot slower than thread context switching.

2.1.4 Concurrency vs Parallelism

When two threads are now scheduled at the same time on different cores, they are executing in *parallel*. However, we say that two threads are running *concurrently*, when their lifetimes overlap. Two threads can run concurrently without ever being scheduled at the same time.
It is easy to show this on a single-core CPU. Imagine the OS only has two threads and keeps context switching between them. These threads run concurrently without ever actually executing instructions at the same time.

Imagine a scenario again with two threads and a single core. The OS can decide to schedule the first thread until it completes and dies. It then schedules the second thread until it completes its tasks and dies. Even though the actual execution time of the first and second thread do not overlap, we still say that they run *concurrently*, because their lifetime overlaps (the second thread was *runnable*, but not scheduled while the first thread was scheduled). We see that concurrency is more of a conceptual property of a program; multiple tasks/threads are order-independent and *can* be run in parallel.

On a high level, we could say that concurrency is about *dealing* with multiple things at the same time, while parallelism is about *doing* these things at the same time.

2.2 Java Threads

The concept of a thread exists in multiple layers of abstraction. What we talked about so far are threads on the OS level. These are used as a means of organising running programs on the system.

Programs themselves can also expose threads to the programmer. In many languages, these are so-called *virtual threads* or *green threads* and have to be treated as an additional level of abstraction, since they are managed by the language itself and not the OS.
In Java however, the thread library exposed by the language is very closely related to actual OS threads, since user created threads are 1-to-1 mapped to OS threads. This is called *native threading*. Technically, the JVM specification does not specify a Java thread to OS thread mapping, but all current JVM

implementations use such a 1-to-1 mapping.
Thus, we can talk about Java threads on the same abstraction level we did so far. Every executing program in Java is made up of threads. Even a normal single-threaded Java program runs on the *main thread*.
We can imagine a Java thread as a sequence of [machine] instructions that are executed sequentially. Keep in mind that the machine will execute machine instructions compiled from your Java code and not your Java code itself.

2.2.1 Creating Java Threads

We demonstrate three viable methods of creating Java threads from within a Java program. When starting such an object, the JVM will tell the OS to create a new thread within the process of the running Java program. Appreciate that now its run method will be executing concurrently to the main thread in its own OS thread.

1. Create a custom class implementing the runnable interface.

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        /* write code to be executed upon
        starting the thread here
        */
    }
}
```

   Now we can instantiate this class in our code, construct a Thread instance with it and have its run method execute concurrently on another thread:

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start(); // starts t and executes its run() method
```

   We start a thread by calling start() on an object instance. Note that we can also execute the run() method of a Runnable or Thread by simply doing a function call:

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.run(); // usually not what we want
```

   This executes the run method on the same thread instead of creating a new one and is usually not what we want. Keep in mind that this is a normal Java class and we could add other methods, fields and constructors. The only limitation is that the run method cannot return anything. We can get around that with shared variables (remember, threads within the same process see the same memory).

2. Extend the Java Thread class:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        /* write code to be executed upon
        starting the thread here
        */
    }
}
```

   Note that using this method, our MyThread class cannot extend another class, while using the first method this would be possible.
   We can now instantiate the MyThread class and start the instantiation.

```
MyThread t = new MyThread();
t.start(); // starts t and executes its run() method
```

3. The most compact method is to create the code that is run by the thread anonymously and inline:

```
Thread t = new Thread() {
    @Override
    public void run() {
        /* write code to be executed upon
        starting the thread here
        */
    }
};
t.start();
```

The @Override decorator is not required and we will leave it away from now on for the sake of having less cluttered code snippets.

2.2.2 Daemon vs Non-Daemon Threads

Java differentiates daemon threads that are responsible for background tasks like memory management (for example the garbage collector is a daemon thread) and non-daemon threads, which execute the program itself. The main thread is non-daemon and so are all user-created threads by default.
A Java program only terminates once all non-daemon threads die. This means in particular that non-daemon threads can continue to run, even when the main() method returned.
User-created threads are non-daemon by default. In this course, we will exclusively work with non-daemon threads.

2.2.3 Setting of this Course

In this course, we analyze multi-threaded [Java] programs. Where is such a program now located in the context of this chapter?
First of all, we are always on a single machine. This course does not cover *distributed* systems consisting of more than one machine. However, the machine we assume usually has multiple cores.
We are usually within a single process (the process of our executing Java program). When we talk about context switches, we usually mean switching from one Java thread to another. But we are also mindful that the system can schedule other processes and threads in between.
We are almost always talking about non-daemon threads, that is, the main thread and user-created threads. Our Java threads can run in parallel on different cores, but also concurrently on the same core. This is all decided by the scheduler and out of our control.

2.3 Summary

What is important to remember from this section is that threads are primarily a unit of organization for the OS and are what ultimately runs on the CPU. Their lightweight nature is what enables the scheduler to quickly swap them in and out of the CPU and thus creating the illusion of many programs running at the same time. Although Java threads are created by the JVM rather than the OS, they are (usually) 1-to-1 mapped to OS threads and hence we can think about them the same way.

2.4 What is/are Again...

- ...*CPU registers*? Registers are a small set (usually about 8-32) of memory locations directly on a CPU core. We cannot access them in a high-level programming language. They are used by the compiler for local variables, as compared to data structures, which are usually placed in the heap. Since registers are directly on the CPU core, they are orders of magnitude faster to access than main memory and even caches.
  The registers used for storing variables are referred to as *general purpose registers*. There are also some special registers, like the instruction pointer (stores the memory address of the instruction to be executed next) and stack pointer (stores the memory address of the top of the stack).

- ...*the heap*? The heap is a contiguous memory region in main memory. Each process has its own heap and uses it for storing datastructures. If you create an array or an object in your Java code, it will be stored on the heap when the program runs. The heap is actively managed by the system to make sure the space is efficiently used.

- ...*the stack*? The stack is also a contiguous memory region in main memory. Usually, each thread within a process has its own stack. It is not actively managed by the system and thus faster to allocate and deallocate memory from than the heap. The compiler usually puts variables on the stack when all registers are already used.

- ...*an address space*? We mentioned that each process has its own address space. What does this mean? To access a memory location, a process does not use the actual *physical* address, but instead

a *virtual* address. We can imagine a virtual layer on top of the memory locations. Each virtual address gets mapped to a physical location when it is used. This means that different processes use a different *virtual* address to refer to the same *physical* address. The advantage of this is that for each process, we can simply slap a virtual layer on top of the memory addresses and to the process it will appear that the memory is empty, since its virtual addresses are not mapped yet. This is why a process cannot access another process' memory: It simply cannot see it.
This whole concept is known as *virtual memory* and it not part of this course. Just know that when we say a new process gets its own address space, it means that the OS initializes a new empty virtual layer to be filled by the process.

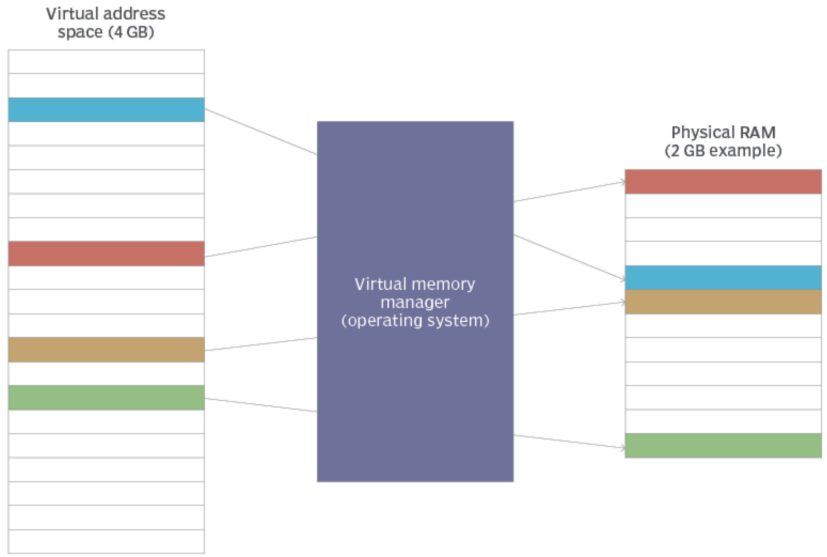


Fig. 5: Virtual to Physical Address Mapping. https://www.techtarget.com/whatis/definition/virtual-address

3 Thread Communication

Now that we discussed what a thread is and how parallelism and concurrency are achieved at the OS level, we are ready to talk about how to organize parallelism.
In this section we take for granted that we have a Java program with multiple threads running *concurrently* and discuss issues that arise and how to solve them when programming in this manner.

3.1 A Simple Multi-Threaded Program: Interleavings

Let us now write our first multi-threaded Java program. We want to write a method that takes two int arrays as an argument, sorts them and then prints the smallest elements of both arrays to the console. Since we know how to create a thread in Java, this is a simple task:

```
public void printSmallest(int[] a, int[] b) {
    /* We first create a Thread object sorting the first array.
    * Then, the main thread sorts the second array and prints to the console. */
    Thread t = new Thread() {
        public void run() {
            // This code is executed when the thread is started.
            Arrays.sort(a);
        }
      };
    t.start();
    Arrays.sort(b);
    System.out.println(a[0] + " " + b[0]);
}
```

We might expect the method to print the smallest element of both arrays. However, when running the program multiple times, we see that this is not always the case. This is because the main thread and the created thread are running concurrently. As soon as t is started, we have two threads running concurrently, which we can imagine as two [sequential] Java programs running at the same time:

```
Main Thread:                                    Helper Thread t:

Arrays.sort(a);                                 Arrays.sort(b);
System.out.println(a[0] + " " + b[0]);
```

It can occur that the main thread executes the println statement before the created thread can sort the array a. This now all depends on how the OS schedules these threads.
We call the (actual real-time) order that instructions of multiple threads are executed in **interleavings**. In this case, there are interleavings where the main thread prints before the other thread finishes sorting the array. Since this is not what the program is intended to do, we call this a *bad interleaving*.

Note that interleavings do not necessarily refer to the interleaving of Java instructions, but rather the interleaving of the machine instructions that are compiled from the Java code.
We will soon discuss solutions to the problem (which is waiting for the helper thread to finish in this case) and other approaches to control the execution of our Java threads. But first, we have to talk about thread states in Java.

3.2 Thread States

Even though we said in the previous chapter that Java threads are 1-to-1 mapped to OS threads, we can still control our Java threads from within our program. The Java Thread library exposes many tools for managing running threads to the programmer. For example do we have various methods that cause a thread not to be scheduled by the OS. To be able to talk about the effects of different thread operations, we need some notion of *thread states*. A Java thread typically goes through the following states:

- *New*: Once the Thread object is created, the thread enters the new state. At this point, the thread is just an object in the heap and no resources have been allocated for it.
- *Runnable*: Once we call `start()` on the new thread object, the system allocates resources to enable its execution and the thread becomes eligible for being scheduled.
- *Running*: When the thread is actually scheduled to run.
- *Not runnable*: We use this as an umbrella state for multiple states that cause the thread to not be runnable. The thread enters this state when one of the following events occur:

1. The `sleep()` method is called to suspend the thread for a specified amount of time to yield control to the other threads.
2. The `wait()` method is called to wait for a specific condition to be satisfied.
3. The thread is blocked and waiting for an I/O operation to be completed. Attempting to acquire a lock and calling `join()` will also cause a thread to be blocked, but these two topics will be covered later on.

These methods and terms will be covered in this chapter. At this point we just note that even though our Java threads are scheduled by the OS, we still have *some* control over them from the language itself.

- *Terminated*: A thread transitions to terminated state when the `run()` method terminates and exits.

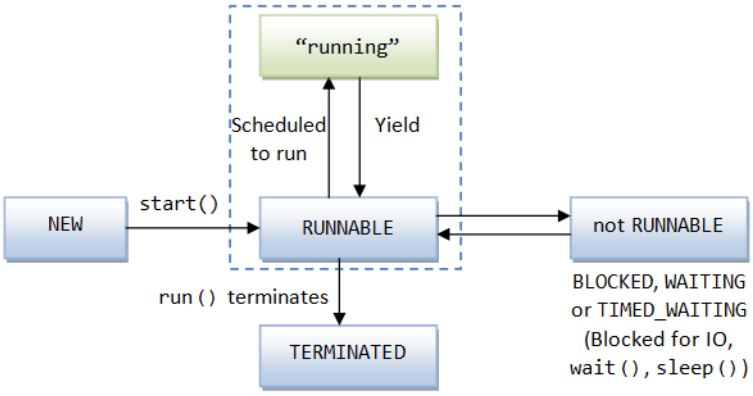


Fig. 7: Java Thread States: https://www3.ntu.edu.sg/home/ehchua/programming/java/j5e_multithreading.html

There is a `getState()` method in Java returning one of the states *new*, *runnable*, *blocked*, *waiting*, *timed waiting* and *terminated*:

```
Thread t = new Thread() {
    public void run() {
        System.out.println(Thread.currentThread().getState());
    }
  };
t.start();
```

```
> RUNNABLE // prints the state to the console
```

A list and short description of each state can be found in the official Java documentation. We notice that Java differentiates some states we simply put into a *not runnable* state. These roughly correspond to the three reasons we mentioned for entering a not runnable state. We need to introduce a few more topics before we can differentiate these properly. For the moment, just appreciate that we have the power of putting our running Java Thread objects into a certain state resulting in the underlying OS thread not getting scheduled.

3.3 Waiting for Another Thread

3.3.1 Busy Waiting

Consider the program from 3.1 again. Our issue was that the main thread did not wait until the other thread completed. A solution to this is to check in a while-loop if the other thread already terminated. With our new knowledge about Java thread states, we can check the state of `Thread t` using the `getState()` method. When the while condition fails, we can finally print:

```
public void printSmallest(int[] a, int[] b) {
    Thread t = new Thread() {
        public void run() {
            Arrays.sort(a);
        }
      };
```

```
    t.start();
    Arrays.sort(b);
    while(t.getState() != Thread.State.TERMINATED) {
        // wait for t to terminate
        System.out.println("Waiting...");
    }
    System.out.println(a[0] + " " + b[0]);
}

> Waiting... // example output of this program to the console
  Waiting...
  Waiting...
  Waiting...
  1 1 // 1 is the smallest element of both arrays.
```

This program is expected to work in *most* cases. However, there are two things wrong with it:

- The first concerns the `getState()` method. We introduced thread states in the previous section and checking such a condition makes sense at first. However, the Java documentation itself states that the method is designed for *monitoring the system state* and not for concurrency control. This means that Java does not guarantee that the returned state is correct and hence we consider the program to not be correct.
- While the helper thread is still running, the main thread keeps checking its state in a while loop. This is referred to as *busy waiting*. The thread is not performing meaningful work. However, the OS does not know this and will still schedule this thread, using up CPU time. Ideally, we would tell the OS that it should not schedule the main thread as long as the other thread is running. This is achieved by the `join()` method introduced in the next section.

3.3.2 join()

The `join()` method does exactly what we were looking for in our example program. When we call `t.join()` in the example, the main thread (which is the one executing the statement) pauses its execution until `t` terminates. We can replace the while loop with a call to `t.join()`.

```
public void printSmallest(int[] a, int[] b) {
    Thread t = new Thread() {
        public void run() {
            Arrays.sort(a);
        }
      };
    t.start();
    Arrays.sort(b);
    try {
        t.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(a[0] + " " + b[0]);
}
```

The program now works as expected and there are no bad interleavings possible anymore. Note that calls to `join()` have to be put into a try-catch block because `join()` can throw an `InterruptedException`, which we need to handle.

Now think about what would happen if the main thread would first call `t.join()` and only then sort the array:

```
t.start();
t.join(); // in real code, this should be in a try-catch block
Arrays.sort(b)
```

Solution: This code still works, but it is slower. This is because as soon as the main thread calls `t.join()` it will not be scheduled anymore until `t` terminates. Only then will it sort the array 'b'. This means that first, array 'a' is sorted and then array 'b', which is no improvement over sequential code.

3.3.3 Busy Waiting vs Joining

We mentioned the drawbacks of busy waiting (waiting thread gets scheduled, taking away CPU time). However, joining is not always better. Calling `join()` can result in a context switch: The thread will be descheduled and some other thread will be scheduled in its stead (maybe the helper thread - but maybe a different one altogether).
When this context switch takes more time than the work of the joined thread, we would have been better off just busy waiting. So, for a very short-lived thread, it may be more efficient to simply busy wait than to call `join()`. However, in almost all cases, calling `join()` will be better.

3.4 Interrupting a Thread

3.4.1 interrupt() on a Runnable Thread

Assume that `t` is a Java Thread object in the following.
When we want to prematurely stop `t`, we can do so by calling `t.interrupt()`:

```
public void interruptThread() {
    Thread t = new Thread() {
        public void run() {
            work() // Thread performs some work
            System.out.println("Finished!");
        }
    };
    t.start();
    t.interrupt(); // interrupt the thread
}

> Finished! // console output
```

When we run this program, nothing seems to happen. Even though t is interrupted, it finishes its work and prints happily to the console.
This is because interruption in Java is implemented with a flag (a flag is just a boolean value). We can imagine that each `Thread` object in Java has a boolean attribute called 'interrupted'. A thread can set the flag of another by calling `interrupt()` on it. The interrupted Thread object can decide itself what happens upon having its flag set.
The `isInterrupted()` method returns whether the current thread has its interrupted flag set. Let us check in the `t.run()` method if t is interrupted:

```
public void interruptThread() {
    Thread t = new Thread() {
        public void run() {
            System.out.println(this.isInterrupted());
            work() // Thread performs some work
            System.out.println("Finished!");
        }
      };
    t.start();
    t.interrupt(); // interrupt the thread
}

> true // t has its interrupted flag set to true
  Finished!
```

Note that an interleaving where `t` checks its interrupted flag before the main thread sets it is also possible in this case. So, 'false' would also be a possible output of this program.

3.4.2 Reacting to an Interrupt

The `Thread` object can now react to being interrupted. Usually when we interrupt a thread, we want it to stop executing. We can implement this for example by checking the interrupted flag and returning from the method if it is set:

```
public void interruptThread() {
    Thread t = new Thread() {
        public void run() {
```

```
            if(this.isInterrupted()) { // also works without 'this' keyword
                return;
            }
            work() // Thread performs some work
            System.out.println("Finished!");
        }
    };
    t.start();
    t.interrupt(); // interrupt the thread
}
```

Now there is no console output since the run method returned after t was interrupted. Again note that there are also interleavings possible where t checks its interrupt flag before the main thread can set it. In such a case, t would still do its work and print to the console.

3.4.3 Interrupting a `Thread` in a `join()` Call

In the previous section, we saw that `join()` can throw an `InterruptedException`. Let us see what happens when we interrupt a thread that just called `join()` on some other thread.

We do the following: First the main thread start two `Thread` objects `t1` and `t2`. In its `run()` method, `t2` joins `t1` by calling `t1.join()`. Now, the main thread interrupts `t2`. Think about what is going to happen:

```
public void interruptThread() {
    Thread t1 = new Thread() {
        public void run() {
            work();
        }
      };
    Thread t2 = new Thread() {
        public void run() {
            work();
            try {
                t1.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
      };
  t1.start();
  t2.start();
  t2.interrupt();
}
```

There are two cases:

- Either `t2` is still in its `t1.join()` call, when the main thread calls `t2.interrupt()`. Remember the try-catch block we had to wrap our `join()` calls in? This is the reason. When a thread is in a `join()` call (meaning that its waiting for another thread to terminate) and its interrupted flag is set to true by some other thread it will throw an `InterruptedException`.
  There are also other methods that automatically throw this exception when the executing thread gets interrupted during the call, which we will see later.
- There are also interleavings where `t2` already terminates before being interrupted by the main thread or `t2` is still executing `work()` when being interrupted. In these cases, nothing happens since the `interrupt()` call will simply raise the interrupt flag of `t2`.

In the first case, an output will look something like this:

```
> java.lang.InterruptedException
        at java.base/java.lang.Object.wait0(Native Method)
        at java.base/java.lang.Object.wait(Object.java:366)
        at java.base/java.lang.Thread.join(Thread.java:2151)
        at java.base/java.lang.Thread.join(Thread.java:2227)
        at interrupt$2.run(interrupt.java:14)
```

We now have a solid understanding of interrupts in Java. We will not use them often throughout this course, but it is important to understand why some methods need to catch `InterruptedExceptions`. What we need to remember are the two different cases: If we interrupt a thread that is currently calling a method which throws `InterruptedException` (`join()`, `wait()`, `sleep()`), it **will** throw such an exception. In any other case, all that happens is that the interrupted flag of the thread is raised and it is our responsibility to write code that checks this flag and reacts appropriately (if we desire such behaviour).

On the OS level, there is also a concept of interrupts, but this has nothing to do with Java interrupts. A Java interrupt can only be caused from within Java. This means that even though the OS can do things like force stop a thread, such things will not throw an `InterruptedException`.

3.5 Synchronize

3.5.1 Critical Sections and Race Conditions

In this chapter, we already talked about interleavings and more specifically, bad interleavings. A correct parallel program needs to guarantee that no interleavings leading to a wrong result are possible.

A *race condition* occurs when the correct execution of a program depends on the real-time execution order (which is dictated by the scheduler). Hence, a race condition occurs when bad interleavings are possible. We can imagine that threads race against each other to execute the critical instructions first. For example our first program in 1.1 suffered from a race condition; the main thread raced to print the smallest elements and the helper thread raced to finish sorting the array.

Let us look at another example, where multiple threads increment a shared counter. We create an array of 10 `Thread` objects, start and join them all and then print the value of the shared counter:

```
class ParallelSum {
    private int counter = 0;

    public int sum() {
        Thread[] threads = new Thread[10];
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread() {
                    public void run() {
                        for(int i = 0; i < 1000; i++)  {
                            inc();
                        }
                    }
                };
            threads[i] = t;
            t.start();
        }
        for(Thread t : threads) {
            t.join(); // this would need to be in a try-catch block
        }
        return counter;
    }

    private void inc() {
        counter++;
    }
}
```

Note that the threads can access the counter because it is a class attribute. If we would declare the counter inside the sum method, the threads could not access it.
When we run the `sum()` method, the returned counter is usually less than the expected 10'000. This is because incrementing the counter in the `inc()` method creates a race condition.

Although the increment is a single Java instruction, the corresponding machine code usually has more than one instruction. The simple counter increment gets compiled to the following Java bytecode:

```
aload_0
dup
getfield      #7
```

```
iconst_1
iadd
putfield      #7
```

We can now see that there are many interleavings possible where multiple threads read the same initial value n (aload), then perform the other instructions in some interleaved order and finally write back (putfield) the same value n+1. In this manner, increments can 'get lost'.
We keep in mind that the machine does not actually execute bytecode (this gets interpreted into machine code which then actually runs on the machine), but we can expect that similar instructions are actually executed on the machine.

We have identified that the increment instruction is a *critical section*. A critical section is a piece of code that only a *single* thread can execute at the same time to guarantee correct (parallel) execution.

3.5.2 Concept of a Lock

3.5.2.1 Locks Provide Mutual Exclusion:

Once we identified a critical section, we must ensure that only one thread can enter it at the same time. This property of the critical section that we are seeking is referred to as *mutual exclusion*. Java allows us to do this using *locks* (although locks are a general concept used in many programming languages, not just Java).
We can imagine that upon entering a critical section, we need to lock it such that no one else can enter. When the section is over, we unlock it again such that other threads can enter. The logic is shown in the following pseudocode:

```
Object lock = new Object();
lock.lock();
// insert critical section code here
lock.unlock();
```

Java provides us the synchronized keyword for this. The equivalent code in actual Java is:

```
Object lock = new Object();
synchronized(lock) {
    // insert critical section code here
}
```

We can neatly wrap the critical section in a synchronized block.

3.5.2.2 Intrinsic Lock or Monitor

Note that the synchronized block always requires an object as an argument. It might seem alien, but in Java we can use *any* object as a lock. We say that each object in Java has an *intrinsic lock*. There is no real intuition behind why this is the case. It is just a convenient implementation decision the language designers made. The intrinsic lock is also called a *monitor lock* or simply *monitor*. In Java, there are also external locks available, which we will see later. But for our current needs, the intrinsic object locks are enough.

In our ParallelSum code, the critical section was the increment, which means we only have to wrap it in a synchronized block. We also do not need to specifically create an object to lock on; we can simply use the ParallelSum object instance - the 'this' object.

```
public void inc() {
    synchronized(this) {
        counter++;
    }
}
```

When the whole method body is a synchronized block, we can instead also define the method itself with the synchronized keyword. So, completely equivalently:

```
public synchronized void inc() {
    counter++;
}
```

Note that when we annotate a method with synchronized, the 'this' object is automatically used to lock on. When we want to lock on another object, we have to use a synchronized block instead.
Now, each time a thread calls the `inc()` method, it has to obtain the intrinsic lock of the ParallelSum object instance. This guarantees us that always only one thread is executing an increment. Even when a thread gets descheduled while incrementing (which can happen), it will simply hold the lock and no other thread can obtain it until it releases it again upon exiting the critical section.

3.5.3 Synchronized Static Methods

Consider a method with the following definition within the class MyClass:

```
static synchronized void myMethod() {
    // method body
}
```

For a non-static method, we said that this is equivalent to having the whole method body in a synchronized(this) block. However, a static method is not associated with an object instance, but with the class itself, so there is no 'this' object. Instead, the method locks on the MyClass.class object. Hence, the code is equivalent to:

```
static void myMethod() {
    synchronized(MyClass.class) {
        // method body
    }
}
```

Consider the following class:

```
class MyClass {
    static synchronized void myMethod() {
        // method body
    }

    synchronized void myOtherMethod() {
        // method body
    }

    synchronized void myThirdMethod() {
        // method body
    }
}
```

Think about which of these methods can be accessed at the same time by different threads. Consider the following main method:

```
public static void main(String[] args) {
    MyClass c = new MyClass();
    Thread t1 = new SomeThreadClass(c); // give c to the constructor
    Thread t2 = new OtherThreadClass(c); // give c to the constructor
    t1.start(); // t1 calls c.myOtherMethod() in its code
    t2.start(); // what methods of c can t2 call at the same time?
}
```

Assume that `t1` calls c.myOtherMethod(). Can `t2` now call `c.myMethod()`? What about `c.myThirdMethod()`?

Solution: The thread `t1` holds the monitor of c (since myOtherMethod synchronizes on the 'this' object). Thread `t2` cannot access `c.myThirdMethod()` now, since to do so, it first needs to obtain the same monitor. However, it can call `myMethod()`, since the monitor of the MyClass.class object is not currently in use.

Although the synchronized keyword is a good abstraction for the underlying locking, we need to pay attention to what monitors are actually obtained when entering such a block/method.

3.5.4 Reentrant Property of Monitors

We said that when a thread holds the monitor of an object, no other thread can obtain it. But what about itself? Consider the following code:

```
public synchronized void method() {
    method2();
}

public synchronized void method2() {
    // some code
}
```

Both methods are part of the same class. When a thread now calls `method()`, it needs to also call `method2()` and obtain the monitor of the 'this' object - which it already has. Thankfully, this is allowed in Java. A thread can obtain a lock an arbitrary number of times. But to release it again, the thread needs to release it just as many times as it acquired it. Hence we say that object monitors in Java are *reentrant*.

3.5.5 Case Study: Summing up Values in an Array

Now we have a more realistic task: We need to sum up the elements of a large array. Since we have a multi-core CPU, we want to do this using a configurable number of threads.

The idea is that with n threads, the i'th thread is responsible for adding up indices i, n+1, 2n+1, 3n+1 and so on. We could also distribute the array indices differently on the threads, but this approach is relatively straightforward.
We start by constructing a class ParallelSum:

```
class ParallelSum {
    private int[] arr;
    private int sum = 0;
    private int numThreads;

    public int sum(int[] arr, int numThreads) {
        this.arr = arr;
        this.numThreads = numThreads;
        Thread[] threads = new Thread[numThreads];
        for(int i = 0; i < numThreads; i++) {
            Thread t = new SumThread(i);
            threads[i] = t;
            t.start();
        }
        return sum;
    }
}
```

```
class SumThread extends Thread {
    private int id;
    public SumThread(int id) {
        this.id = id;
    }
    public void run() {
    }
}
```

Fig. 8: Skeleton of a ParallelSum class.

The class does not have a constructor. Instead, we pass the array `arr` we want to sum and the desired number of threads as arguments to the sum method. All we are doing at this point is saving `arr` and `numThreads` to class attributes and creating an array of `Thread` objects.
The `Thread` class we created does not yet do anything. But we already gave the `Thread` objects IDs from `0` to `numThreads - 1`. You can think for yourself how that could help us implement the desired pattern.

Next, we need to actually implement the run method of the threads. We want each thread to sum up its indices locally and then add this sum to the global sum. To do so, we implement a sumUp method, which takes the ID of the `Thread` object (the one we gave it in the constructor) as an argument and sums the corresponding indices. Another `increaseSum()` method increments the counter by a specified amount:

```
class ParallelSum {
    private int[] arr;
    private int sum = 0;
    private int numThreads;

    public int sum(int[] arr, int numThreads) {
        this.arr = arr;
        this.numThreads = numThreads;
        Thread[] threads = new Thread[numThreads];
        for(int i = 0; i < numThreads; i++) {
            Thread t = new SumThread(i);
            threads[i] = t;
            t.start();
        }
        return sum;
    }

    private int sumUp(int id) {
        int localSum = 0;
        for(int i = id; i < arr.length; i += numThreads) {
            localSum += arr[i];
        }
        return localSum;
    }

    private void increaseSum(int inc) {
        sum += inc;
    }
}
```

```
class SumThread extends Thread {
    private int id;
    public SumThread(int id) {
        this.id = id;
    }
    public void run() {
        int inc = sumUp(this.id);
        increaseSum(inc);
    }
}
```

Fig. 9: Flawed implementation of ParallelSum.

Does the sum method work how we intend it to? Are there any race conditions or other concurrency bugs? Try to find mistakes in the code before reading on.

The first thing you should notice is that the threads are not joined. The main thread can return the sum before all helper threads finished their work. This is a race condition.

The next problem is that writes to the sum variable also create a race condition. Think about how we can use the synchronize here. What are the critical sections?

Solution:

```
class ParallelSum {
    private int[] arr;
    private int sum = 0;
    private int numThreads;

    public int sum(int[] arr, int numThreads) {
        this.arr = arr;
        this.numThreads = numThreads;
        Thread[] threads = new Thread[numThreads];
        for(int i = 0; i < numThreads; i++) {
            Thread t = new SumThread(i);
            threads[i] = t;
            t.start();
        }

        for(Thread t : threads) {
          t.join(); // this would have to be in a try-catch block
        }
        return sum;
    }

    private int sumUp(int id) {
        int localSum = 0;
        for(int i = id; i < arr.length; i += numThreads) {
            localSum += arr[i];
        }
        return localSum;
    }

    private void synchronized increaseSum(int inc) {
        sum += inc;
    }
}
```

```
class SumThread extends Thread {
    private int id;
    public SumThread(int id) {
        this.id = id;
    }
    public void run() {
        int inc = sumUp(this.id);
        increaseSum(inc);
    }
}
```

We only need to synchronize on the increaseSum method, since it is the only critical section. For the sumUp method, each thread has its own localSum variable and there are no concurrent accesses to any data. This code now works as expected.
Note that for the code to work, we define the SumThread class as a nested class of ParallelSum, since this way the SumThread instances can access the methods and fields of the ParallelSum class. If SumThread is defined externally, this approach does not work.

3.6 Producer-Consumer Scenario: Wait/Notify

3.6.1 Problems with Synchronized

The next issue we want to tackle is a producer-consumer scenario. This occurs for example if we have a datastructure some threads are writing to and others are reading from. We know by now that when multiple threads access the same data, we need to properly synchronize in order to prevent race conditions.

Consider following two thread classes; a producer and a consumer class:

```
public class Consumer extends Thread {
    private final UnboundedBuffer buffer;

    ...
    public void run() {
        while(true) {
            while(buffer.isEmpty());
            computation(buffer.remove());
        }
    }
}
```

```
public class Producer extends Thread {
    private final UnboundedBuffer buffer;

    ...
    public void run() {
        while(true) {
            int prime = computePrime(prime);
            buffer.add(prime);
        }
    }
}
```

Both operate on the same datastructure - a shared buffer which we assume to be unbounded (the producer can add an arbitrary amount of elements). The producer keeps adding elements while the consumer keeps reading them (and perform a long computation with them). The problem is that the consumer can only read if the buffer is not empty. Hence, it checks the condition in a while loop before accessing. Can you see the problem?

3.6.1.1 Problem: Race Condition

Assume we have two consumer threads A and B both executing their `run()` method and the buffer containing exactly one element:

`Thread A:`

```
0: while(buffer.isEmpty());
1: computation(buffer.remove());
```

`Thread B:`

```
2: while(buffer.isEmpty());
3: computation(buffer.remove());
```

Consider the interleaving of instructions 0,2,1,3. Both threads see that the buffer is not empty. Thread A removes the element first and executes its computation. Thread B also tries to remove, but the buffer is now empty. This will throw an exception. Hence, the while loop checking whether the buffer is empty is a critical section and needs to provide mutual exclusion. Think about which code blocks we need to synchronize.

3.6.1.2 Problem: Concurrent Remove/Add

We start by simply sychronizing the while(buffer.isEmpty()) loop. Consider now a consumer thread A executing the synchronized block. We are guaranteed that no other consumer thread can try to remove elements from the buffer at the same time due to the synchronization. But a producer thread B can add to the buffer at the same time:

`Consumer Thread A:`

```
computation(buffer.remove());
```

`Producer Thread B:`

```
buffer.add(prime)
```

We do not know the implementation details of the underlying `UnboundedBuffer` datastructure, but we can assume that it is implemented in a non-trivial way, meaning that an add and remove are compiled to many machine code instructions that can be interleaved in ways that corrupt the datastructure. For example if the datastructure is implemented as a linked list, we can imagine interleavings of the `add()` and `remove()` methods where pointers of the list elements are messed up. To make sure nothing bad can happen, we also synchronize the `buffer.add(prime)` instruction on the buffer monitor. Our final solution using synchronization is the following:

```
public class Consumer extends Thread {
    private final UnboundedBuffer buffer;

    ...
    public void run() {
        while(true) {
            synchronized(buffer) {
                while(buffer.isEmpty());
                computation(buffer.remove());
            }
        }
    }
}
```

```
public class Producer extends Thread {
    private final UnboundedBuffer buffer;

    ...
    public void run() {
        while(true) {
            int prime = computePrime(prime);
            synchronized(buffer) {
                buffer.add(prime);
            }
        }
    }
}
```

Now, there are no concurrent accesses to the buffer. So it is all good, right? See if you can spot anything that could go wrong.

3.6.1.3 Problem: Only synchronize where necessary:

Another problem with the code is that the synchronized block of the Consumer thread class includes the computation with the removed element. This means that no other thread can add to or remove from the buffer while this Consumer thread object computes. We must make sure to only synchronize where absolutely necessary, because we lose parallelization by using synchronized. Hence we write the removed element into an int and compute outside of the synchronized block:

```
public void run() {
    while(true) {
        synchronized(buffer) {
            while(buffer.isEmpty());
            int prime = buffer.remove();

        }
        computation(prime);
    }
}
```

3.6.1.4 Problem: Deadlock

Imagine we start with an empty buffer. A consumer thread (let us call it thread A) is now started, obtains the monitor of the buffer object and busy waits for elements to be added to the buffer. Do you see the problem? A producer thread B is now started and wants to add an element to the buffer. But to do so, it needs to obtain the lock for the monitor. This is held by thread A busy waiting for B. We reached a *deadlock*. This is a common bug when using locks in our programs. We can define a deadlock as a circular blocking between threads, such that no thread can make progress anymore. Such a state is clearly reached here.

3.6.2 Wait/Notify

To resolve this situation, we need some way for the consumer thread to give up its lock temporarily until the condition it is waiting for is fulfilled. This kind of communication between threads is possible in Java with the `wait()` and `notify()` methods.

3.6.2.1 We call `wait()` on a monitor:

All we need to do to resolve above situation is tell the consumer thread to **wait** while the buffer is empty. What are we waiting for exactly? Remember that we use *wait()* to give up the monitor of the buffer object, such that other threads can make progress. We always call `wait()` on a *monitor*. A monitor we are currently the owner of, that is. For example in a synchronized block (or method). We write:

```
synchronized(buffer) {
    while(buffer.isEmpty()) {
        try {
            buffer.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    int prime = buffer.remove();
}
```

Like for `join()`, calls to `wait()` have to be put into a try-catch block, since `wait()` can throw an `InterruptedException`. Like for `join()`, we keep this in mind, but leave away the try-catch block from now on for the sake of having more concise code throughout the script.

```
synchronized(buffer) {
    while(buffer.isEmpty()) {
            buffer.wait(); // this would have to be in a try-catch block
        }
    }
    int prime = buffer.remove();
}
```

3.6.2.2 What are we waiting for? Notify:

For how long does the thread wait now? It waits until another thread calls `notify()` on the same monitor. In this example, the producer threads need to call `notify()` after adding an element to the buffer. Calling `notify()` does not give up the lock. Rather does it wake up a waking thread (more on that in the next paragraph), which will then queue for the lock. However, the notifying thread will keep the lock until it finishes its synchronized block (or gives up the lock otherwise - for example by calling `wait()` itself).
Let us analyze the two critical sections with the wait/notify implementation now:

`Consumer Thread`:

```
synchronized(buffer) {
    while(buffer.isEmpty()) {
        buffer.wait();
    }
    int prime = buffer.remove();
}
```

`Producer Thread`:

```
synchronized(buffer) {
    buffer.add(prime);
    buffer.notify();
}
```

Calls to `notify()` do not have to be placed in a try-catch block.

3.6.2.3 Who does notify wake up?

Who gets woken up by the `notify()` now? Exactly one thread out of the set of threads that have called `wait()` on the corresponding monitor. Let us do a sanity check of the logic of the two critical sections. A consuming thread does the following:

1. Obtain the monitor of the buffer object instance.
2. Check if the buffer is empty. Since the thread owns the monitor, no other thread can interfere and create inconsistencies. If the buffer is empty, call `wait()`, which releases the lock for other threads to take it. If the thread then gets woken up by a `notify()`, it will start over (obtain lock and check if empty).
3. When the buffer finally is not empty, it removes an element. At this point, the thread owns the monitor again. Either the buffer was not empty from the start (thread never gave up the lock) or it was notified by another thread, reobtained the monitor and checked that the buffer is not empty.
4. Release the monitor.

A producing thread does the following:

1. Obtain the monitor of the buffer object instance.
2. Add an element to the buffer. No thread can interfere since the monitor is held.
3. Notify **one** thread that called `wait()` on the buffer monitor. This thread can now try to obtain the monitor to continue its work.
4. Release the monitor. Again note that notify itself does not release the monitor. The monitor is released now because the synchronized block is over here.

The methods are correct now, no bad interleavings are possible anymore.

3.6.3 notifyAll()

We learned that calling `notify()` wakes up a single thread out of the set of threads that called `wait()` on the same monitor. The Java documentation states that the thread woken up by `notify()` is arbitrary. Calling `notify()` hence does not always work as expected, especially since we cannot control which of the threads gets woken up. Imagine in our producer-consumer example, we had a second consumer thread class, which only consumes when the buffer has at least 20 elements:

```
synchronized(buffer) {
    while (buffer.size() < 20) {
        buffer.wait(); // this would have to be in a try-catch block
    }
    int val = 0;
```

```
    for (int i = 0; i < 20; i++) {
        val += buffer.remove();
    }
}
```

If we now call `notify()` from a producing thread, we do not know what kind of consumer thread wakes up. If there are threads of both kinds waiting and one demanding at least 20 elements is woken up, it acquires the monitor, sees that there are less than 20 elements in the buffer and calls `wait()` again. Now a consumer thread (requiring only one element) could perform work, but is still waiting. In this situation, we are hence forced to call `buffer.notifyAll()`. This of course wakes up all threads that previously called `buffer.wait()`.
When there are a lot of threads waiting for a monitor, calling `notifyAll()` is very expensive. All of the waiting threads are woken up and compete for the lock, using up precious CPU time. Hence we call `notifyAll()` only when necessary. Which is when we cannot guarantee that no matter which thread (that called `wait()`) is woken up, its condition will be satisfied.

3.6.4 Always `wait()` in a While-Loop

Do we even need the while-loop to check the condition in our example? Let us replace the while with an if:

```
synchronized(buffer) {
    if (buffer.isEmpty()) {
        buffer.wait();
    }
    int prime = buffer.remove();
}
```

The logic does make sense at first glance. We check if the buffer is empty. If it is, we wait until a producing thread notifies us. Since there is only one `notify()` in our code (directly after a producing thread adds to the buffer) the notified thread should be able to perform its removal without rechecking whether the buffer is empty. However, there are multiple reasons why the condition for a wait must be in a while-loop:

1. *Spurious wakeups*: A waiting thread can get woken up without any `notify()` having occured. If there is no while-loop, the thread now moves on without checking the condition again, which is undesired behaviour. Such events are called spurious wakeups and the reason they occur is buried deep in the OS. An easy answer is that they are hard (and expensive) to prevent. Hence the OS just allows them for performance reasons and also because allowing them does not pose a big inconvenience to the programmer; the conditions should be checked in a while-loop anyways because of the other listed reasons.

2. Imagine two consumer threads (which wait for the buffer to not be empty) are notified by a `buffer.notifyAll()` call. Both queue to obtain the monitor. One goes first, removes the element and releases the monitor. Since only one element was in the buffer, it is now empty again. Now the second thread obtains the monitor and tries to remove an element, which throws an exception since the buffer is empty. Note that the thread does not check the condition `buffer.isEmpty()` again after waking up from the `wait()` call. If we had a while-loop instead of an if-statement, the thread would simply check again if `buffer.isEmpty()`, see that the buffer is, in fact, empty and call `wait()` again.

3. Waiting threads might have different conditions to check. In above example, we had threads waiting for a different number of elements in the buffer. If they do not check their condition again after being notified, they might continue even though their condition is not met (the `notify()` or `notifyAll()` was meant for the condition of another thread).

3.7 Thread States Revisited

At the beginning of this chapter we talked about thread states and specifially about a *not runnable* state. Having introduced monitors, `wait()`/`notify()` and `join()`, we can now further divide this state. Consider the new state diagram.

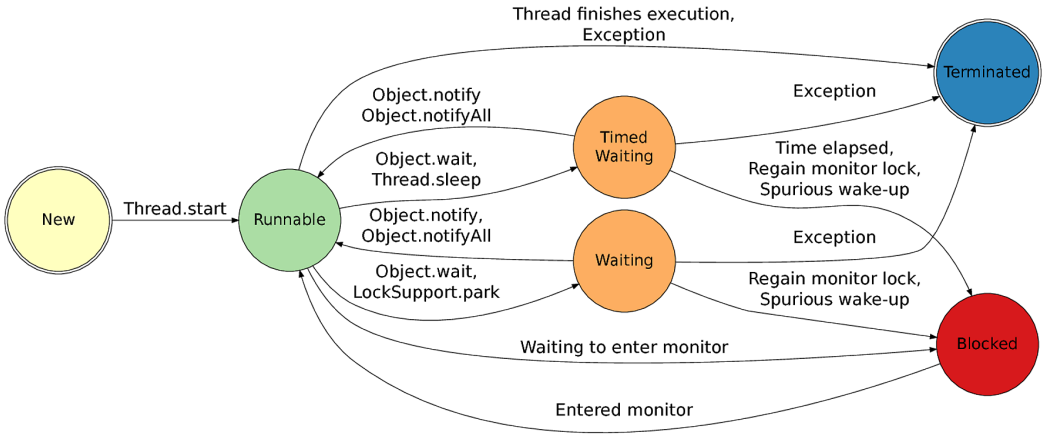


Fig. 16: Java Thread State Diagram. Credit: https://konradreiche.com/

3.7.1 The Timed Waiting State

This is similar in behaviour to the normal waiting state. Methods that cause a transition to the timed waiting state:

- `Object.wait(long timeout)`: This causes the thread to wait on the Object monitor until either the timer elapses or another thread calls `notify()`/`notifyAll()` on the monitor.
- `Thread.sleep(long millis)`: This method causes the Thread object to pause execution for the specified number of milliseconds. Note that this thread cannot be woken up by any `notify()`. The method does not have anything to do with monitors, `wait()` and `notify()`.
- `Thread.join(long millis)`: This causes the calling thread to wait for either the timer to expire or the joined thread to terminate (whichever happens first).

Note that the `Object.wait(long timeout)` and `Object.join(long timeout)` are overloads of the usual `wait()` and `join()` we already know. When we give a `long` as an argument to those methods, the thread will go to the `timed waiting` state instead of a `waiting` state. There are also other methods leading to this transition, but we will not work with them for the remainder of this course. Note that the specified wait times are not guaranteed to be exact, as the underlying OS is responsible for the timing.

3.7.2 Java Thread States vs OS Thread States

A question that might come to mind is whether these states are just defined and managed by Java or if they also exist on the OS level. The answer is that in most operating systems, thread states similar to the Java thread states exist. However, Java does not specify how it maps Java thread states to OS thread states. Java leaves this open to the specific implementation. This makes sense, since different JVMs run on different underlying operating systems, which can differ greatly.

3.7.3 State Transitions in `wait()`/`notify()`

Let us now specifically focus on the thread state transitions occuring in `wait()`/`notify()`:

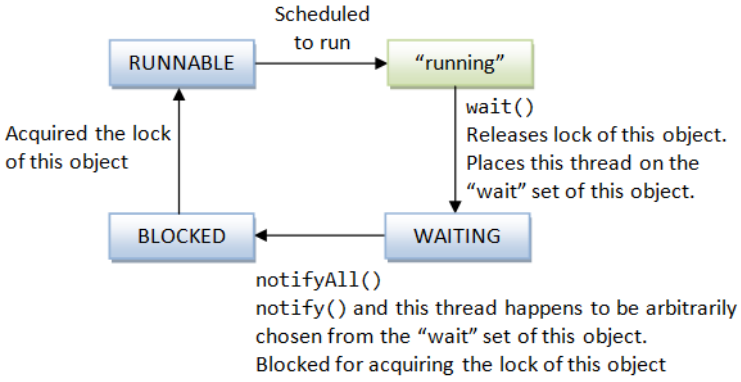


Fig. 17: State Transitions Using `wait()`/`notify()`: https://www3.ntu.edu.sg/home/ehchua/programming/java/j5e_multithreading.html

A running thread (owning a monitor) calling `wait()` will release the monitor and change to a `waiting` state. The waiting thread will not continue executing its code until notified by another thread (except for the discussed rare spurious wakeups). Note here that the thread is in a waiting state on the Java language level. It may or may not actually be transitioned to a waiting state on the OS level. The difference is that if it is in a waiting state on the OS level, it will not be scheduled until notified (or spuriously woken up). But Java could also decide to implement the call to `wait()` as a busy wait (for example), in which case the underlying OS thread is still in some runnable state and scheduled (to execute its busy wait instructions). This mapping between Java and OS thread states is not specified in the Java documentation and is implementation dependent.
Upon being notified, the thread changes to a `blocked` state, because before being runnable, the thread needs to reacquire the monitor. This blocked state can be imagined as the waiting area for the monitor. When the monitor is acquired, the thread goes back to a runnable state and once scheduled will continue with the next instruction after the call to `wait()`.

3.7.4 State Transitions in `join()`

What state does a thread transition to when calling `Thread.join()`? To answer this, let us look at a specific Java implementation of the `join()` method:

```
public final synchronized void join() throws InterruptedException {
    while(isAlive()) {
            wait(0);
    }
}
```

This is the code that is executed when calling `join()` (without specifying a timeout) in OpenJDK (an open-source Java implementation). Suppose that Thread object `t1` calls `t2.join()`. The following happens now (assuming our system uses OpenJDK):

1. The calling thread `t1` acquires the monitor of `t2` (because of the `synchronized` keyword).
2. `t1` checks if `t2` is not yet terminated (using `isAlive()`).
3. `t1` calls `t2.wait()`. This transitions `t1` to a `waiting` state.
4. Upon terminating, `t2` notifies on its monitor, causing `t1` to return from a `waiting` state. You may ask where `notify()` is called. In the OpenJDK implementation, it is called in an internal method, which itself is called when `t2` is terminating.

Hence, the thread calling `join()` transfers into a `waiting` state (this is guaranteed for any Java implementation).
Note that this is just one specific implementation of Java, and there are others, like the OracleJDK. Using `wait()` to implement `join()` is common, but we cannot assume that every Java implementation executes equivalent code.

4 Exploiting Parallelism on a Single Core

In the last chapter, we talked about how to use the Java thread library, how threads can communicate and solve different problems related to organizing concurrency.
Until now, our parallelization approach was the following: We have a sequential program and decide as programmers that some parts can be executed concurrently. So we create a new Java Thread object and let it execute this code part concurrently. What we assume there is that we have multiple *cores* that can execute multiple threads in parallel. This kind of parallelism is exploited by the programmer itself.
However, some clever tricks can also be applied to get parallelism on a *single* core out of a program. These are generally abstracted away from the programmer and require special hardware support.

4.1 Instruction-Level Parallelism (ILP)

ILP refers to parallelism that can be obtained by executing independent instructions in parallel. Consider the following pseudo-assembly:

```
1: ADD R1, a, b // R1 <- a + b
2: ADD R2, c, d // R2 <- c + d
3: ADD R3, R1, R2 // R3 <- R1 + R2
```

The first two instructions are completely independent and can be executed concurrently. The third however has to wait for the first two to complete. We can exploit *ILP* for the first two instructions. Generally, we say that two instructions are independent if the result of one is not the operand (or input) of the other.

4.1.1 ILP vs Multi-Threading

We clearly differentiate ILP from concurrency (or multi-threading): ILP concerns the execution of a *single* thread, while concurrency intrinsically concerns multiple threads. Basically, we can first start multiple threads (multi-threading to exploit multiple cores on the machine) to each execute a sequential part of the program. Then, within each thread (that can itself be viewed as a sequence of instructions), we seek independent instructions that *could* be executed in parallel. Now of course it is inefficient to start another thread to execute a single instruction concurrently (because of thread creation and context switch overhead). Also, assume that we already have enough threads to keep all cores busy.
It would be nice to execute this independent instruction in parallel *within* the core the thread is alredy scheduled on. This kind of parallelism is what ILP refers to. To exploit ILP, we need additional hardware of course; if the core the thread runs on does not provide the hardware to execute multiple instructions in parallel, no software tool will be able to make it happen.
In the following, we will explore different approaches towards exploiting ILP.

4.2 Pipelining

Pipelining is a very general concept; it is about partitioning a process (do not think of an OS process here) into different stages, usually each with its clear function.

4.2.1 A Simple Example: Latency and Throughput

Let us analyze the process of doing laundry. We can divide this into the following stages:

- Washing (15 minutes)
- Drying (10 minutes)
- Folding the dry clothes (5 minutes)
- Put clothes back into closet (5 minutes)

Now imagine we have multiple loads of laundry we need to process. We can wait for one load to complete with all stages and then start with the next. But it would be more efficient to start with the second load as soon as the first is done with washing (the first stage) and so on. We receive the following execution:

Time (m)	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Load 1	w	w	w	d	d	f	c												
Load 2				w	w	w	d	d	f	c									
Load 3							w	w	w	d	d	f	c						
Load 4										w	w	w	d	d	f	c			
Load 5													w	w	w	d	d	f	c

What we created here is a pipeline. Any modern CPU uses pipelining when executing instructions. The

stages within a core might be to (i) fetch the instruction from memory, (ii) decode it (read the instruction and decide what needs to be done), (iii) execute the specified operation (for example add the operands), and (iv) write back the result.
With each stage, an execution unit is associated. In the laundry example, we have a washing machine for the washing stage, a dryer for the drying, a person who needs to fold the clothes and again a person who needs to put the clothes back into the closet. In a CPU, this is the same. In the example, we have a decoder for the first stage, a Load/Store unit for fetching operands, an ALU (Arithmetic Logic Unit) for computing results and so on.
We can now define a few properties of a pipeline.

Definition 4.2.1. Latency is the time it takes to perform a given computation.

Latency is deliberately defined vaguely. Each element of the pipeline has a latency, in this example it is 35 minutes (how long it takes to traverse all stages). Note that latency need not be constant over time and can instead be different for each element. This occurs when there are gaps in the pipeline, i.e. an element has to wait for its predecessor to finish before it can start with the next stage. Then, its latency will be higher than the latency of its predecessor. We will soon see an example where this is the case.
We can also talk about the latency of the whole pipeline to process five elements, which is 95 minutes in this case. We see that latency simply refers to the duration of a given computation and we need to specify what exactly we mean when we talk about latency.

The goal of pipelining is not to improve (i.e. decrease) the latency of each element, but to improve (i.e. increase) the *throughput*. The throughput refers to the amount of work a system can produce in a given amount of time. Our laundry pipeline can (at full pipeline capacity) output one load every 15 minutes, which is its throughput. In a general-purpose CPU, we are interested in both latency and throughput. The latency dictates how long an instruction needs to execute. The throughput dictates how many instructions can be executed by the CPU pipeline in, say, one second. Both are important, but usually sacrificing a bit of latency in order to get more instructions per second (i.e. more throughput) is a good tradeoff. We define throuhgput:

Definition 4.2.2. Throughput is the number of elements that exit the pipeline (at full capacity) per a given time unit. Throughput can be calculated as follows for *any* pipeline with one execution unit per stage:

$$Throughput = \frac{1}{max(computationtime(stages))}$$

In our example, the formula gives the expected 1/15 minutes. We can now clearly quantify the advantage of pipelining the process. The throughput of sequential execution is trivially 1/35. With pipelining, we managed to more than double the throughput, while the latency remained the same.

Note that in our throughput computation, we assume that our pipeline is running at full capacity. But this is not really the case. If we fix the amount of loads we want to process, we can also define the throughput under consideration of lead-in (filling the pipeline) and lead-out (emptying the pipeline) time, i.e. the time where the pipeline is not at full capacity.

Definition 4.2.3. Throughput under consideration of lead-in and lead-out time given n elements traverse the pipeline is the average time it takes to output an element. This throughput can be calculated as follows for *any* pipeline with one execution unit per stage:

$$\frac{n}{overall\ time\ for\ n\ elements}$$

$$= \frac{n}{n * max(computationtime(stages)) + sum(computationtime(all\ stages\ except\ longest))}$$

The second formula might look daunting, but note that the denominator is just equal to the overall time it takes all n elements to finish. The second formula is also easier to compute, since it does not require us to draw out the pipeline (to find the overall execution time). With the second formula, we can also see that when we let n go to infinity, the formula converges to the formula for throughput. This is in line with our expectations, since for n going to infinity, the lead-in and lead-out time are negligible since they are constant.
Considering lead-in and lead-out, the throughput of our laundry pipeline given 5 loads traverse it is 1/19

(we use the first formula with n=5 and *overall time*=95). If we let n grow larger, this will converge to the 1/15 throughput.

4.2.2 Balanced Pipelines

In an effort to improve our throughput, we buy a new washing machine. The new execution time for the washing stage is 5 minutes.

Time (m)	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70
Load 1	w	d	d	f	c										
Load 2		w	-	d	d	f	c								
Load 3			w	-	-	d	d	f	c						
Load 4				w	-	-	-	d	d	f	c				
Load 5					w	-	-	-	-	d	d	f	c		

We see that our new throughput is 1 load per 10 minutes. Latency, however, is not constant anymore and increases with each element. We say that the pipeline is *unbalanced*. We can now guess what *balanced* means.

Definition 4.2.4. A pipeline is called **balanced** if and only if the latency of all elements traversing it is constant. A balanced pipeline does not have any gaps.

This example illustrates one important problem that can occur in pipelines: Despite achieving higher throughput, the pipeline became unbalanced, hence the latency increases indefinitely with every additional load traversing it. We can imagine that this is not desirable in CPUs. Imagine that each instruction takes longer to execute than the previous one. While throughput may be good, the latency will become unbearable over time. Think about how we can solve the problem in this case and make the pipeline balanced again.

Solution: We can artificially increase the time of the washing stage to 10 minutes, for example by simply waiting for another five minutes before advancing to the drying stage. This balances our pipeline at the expense of increasing the latency for the first element. A price we most likely are ready to pay.

But what made our pipeline unbalanced? Before, the first stage (washing) was the longest. But now, there is a stage that takes longer; the drying stage. The loads finish washing too quickly and have to wait for the previous load to finish drying. We can quickly identify whether a pipeline is balanced with the following lemma that is given without proof:

Lemma 4.1. *Equivalently, a pipeline (with one execution unit per stage) is* ***balanced*** *if and only if no stage takes strictly longer than the first one.*

Before, this was the case. No stage was longer than the washing stage initially. Decreasing the washing stage to 10 minutes would also have the pipeline remain balanced, since no stage takes longer than 10 minutes, but anything below 10 minutes for the washing stage makes the pipeline unbalanced as stated by the lemma.

4.2.3 Multiple Execution Units

Our definitions for throughput and our lemma for balance assumed that the pipeline has one execution unit per stage.
Take the first throughput formula. We can use such an easy formula because the longest stage will always be the bottleneck as long as we only have only a single execution unit. This is because all elements will have to wait for the current element to complete the stage. But now imagine in our unbalanced example we had two dryers instead of one. Then, even if a dryer is used, the next load does not have to wait. It can simply use the second dryer. Even though it is the longest stage, drying is not a bottleneck anymore. Let us plot the execution assuming we have a second dryer.

Time (m)	0	5	10	15	20	25	30	35	40	45	50
Load 1	w	d	d	f	c						
Load 2		w	d	d	f	c					
Load 3			w	d	d	f	c				
Load 4				w	d	d	f	c			
Load 5					w	d	d	f	c		

The pipeline is balanced again and the throughput is one element every 5 minutes. We do not have an easy criterion for balance and formula for throughput anymore, we need to draw the pipeline.

In a CPU, we can for example have multiple ALUs or multiple memory units. Assuming decoding is a lot faster than computing a result, these additional units would increase the throughput of the CPU pipeline.

4.2.4 Pipelining as an Approach towards exploiting ILP

Pipelining is a very general concept that is not fundamentally linked to ILP. It is such a general concept that the CPU pipeline was invented before the term ILP was coined even. Still, we can clearly say we use pipelining to exploit ILP.
First of all, pipelining happens on a single core, by partitioning the instruction execution of each core into multiple stages with (at least) one execution unit per stage. This makes sense even without the context of ILP.

However, a CPU pipeline can provide more speedup when there are ILP opportunities in the code. This is because if instructions are dependent, they at least need sufficient distance in the pipeline (or cannot be in the pipeline at the same time at all).
To see this, consider an instruction that has an operand that is the result of a previous instruction. It needs to wait for this instruction to reach a stage in the pipeline where it writes back its result into a register (or memory). Let us say this write-back happens in the 10'th stage. Fetching the operands happens in the second stage. Now, our dependent instruction can reach stage two only after the other instruction completed stage ten. Assuming our pipeline fetches a new instruction in each cycle, we need eight (independent) instructions in between those two dependent ones to keep the pipeline full.
Hence, a CPU pipeline requires careful ordering of the instructions to ensure dependent instructions have enough distance. This can be done both by the compiler cleverly ordering instructions and also the CPU itself detecting dependencies and delaying a dependent instruction. But such implementations are out of the scope of this course.

Modern CPUs on, say, Laptops usually have deep pipelines with dozens of stages. More stages generally means a little more latency, but more throughput (since the length of the longest stage usually decreases with more fine-grained stages). This also means that modern pipelines require a lot of ILP and clever reordering to reach their maximum throughput (since more stages usually means dependent instructions require more distance in the pipeline).

4.3 Vectorization

4.3.1 Vectorization vs Multi-Threading: SIMD

Vectorization is an optimization that is applicable when we have operations on vector-like data in our code. In Java, these datastructures are usually arrays. Consider the following Java code:

```
public double[] arrayMultiply(double[] a, double[] b) {
    final int SIZE = a.length;
    double[] c = new double[SIZE];
    // Compute the element-wise product of a and b
    for (int i = 0; i < SIZE; i++) {
        c[i] = a[i] * b[i];
    }
    return c;
}
```

This loop is obviously parallelizable. All iterations of the loop are independent of each other and could be executed in parallel. This is a special form of ILP, where the independent instructions are all of the same kind - additions on doubles. From what we learned in the last chapter, we can definitely optimize this. Assume we have a four-core CPU. We can now create four Java threads and let each of them compute one fourth of the loop. However, we now want to discuss a different approach that can be done on a *single* core. Because remember that in this chapter, we focus on speeding up a thread of instructions executing on a single core.

4.3.2 SIMD

The approach we talk about is called *SIMD*. Note that there are other possible implementations for exploiting vectorization than the one we are about to introduce. It is however one that is commonly used in modern CPUs.
SIMD stands for *Single Instruction, Multiple Data*. We have an operation (for example a multiplication) and want to apply it to multiple data values at the same time (for example a whole *vector* of values). Let us first look at how such an instruction works normally, without any SIMD. In a simple processor we have

registers and the ALU (Arithmetic Logic Unit) organized as shown in the figure below. Some registers `r1` and `r2` are used as inputs and the result is stored in another register `r3`. Of course any register could be used.

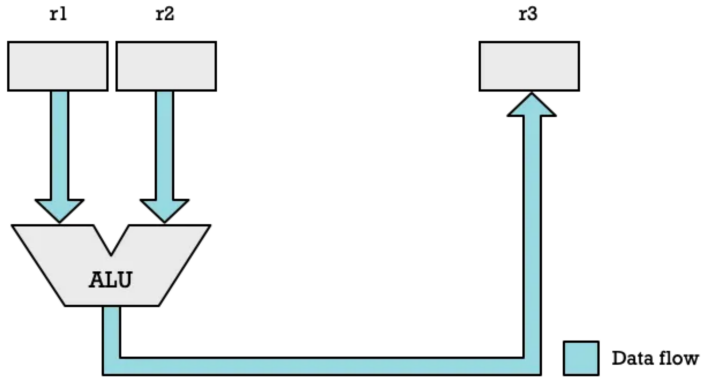


Fig. 18: Normally: Single Instruction, Single Data: https://itnext.io/vector-processing-on-cpus-and-gpus-compared-b1fab24343e6

When the CPU now executes the instructions like this, it reads two doubles into `r1` and `r2` respectively, multiplies them and writes them into `r3`. Then, it goes on to the next instruction and again loads two doubles and so on. In pseudo assembly code, this would look something like this:

`Loop`:

```
result[i] = a[i] * b[i];
```

`Corresponding Pseudo Assembly`:

```
LOAD R1, (a) // load next double of 'a'
LOAD R2, (b) // load next double of 'b'
MUL R3, R1, R2 // R3 <- R1 * R2
```

Now we want to enhance this CPU to enable SIMD instructions. For this, we need to increase the width of the registers and add another ALU. The registers `r1`, `r2` and `r3` we assumed so far are 64 bits wide (they hold exactly one double). Now, we assume we have registers `v1`, `v2` and `v3` that are 128 bits wide each. This means that we can put *two* doubles into each such register. Now we can execute the following:

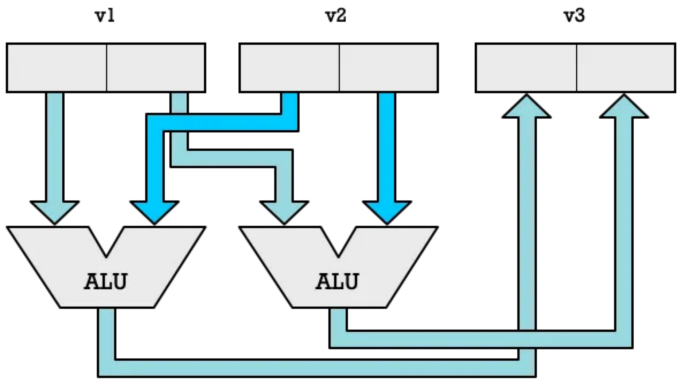


Fig. 20: Single Instruction, Multiple Data: https://itnext.io/vector-processing-on-cpus-and-gpus-compared-b1fab24343e6

In one instruction, we can now load *four* doubles, two into each of `v1` and `v2`. Then compute two multiplications in parallel using the two ALUs and write the two results into the register `v3`. In pseudo assembly code, this would then look something like this:

`Unrolled Loop`:

```
result[i] = a[i] * b[i];
result[i+1] = a[i+1] * b[i+1];
```

`Corresponding Pseudo Assembly`:

```
VLOAD V1, (a) // load next two doubles of 'a'
VLOAD V2, (b) // load next two doubles of 'b'
VMUL V3, V1, V2 // V3 <- V1 * V2
```

We introduced a special VMUL (vector multiply) instruction that uses special vector registers. We now know that these vector registers are just wider than normal registers and the vector instruction just uses multiple ALUs to compute the result in parallel.

This means that we can execute this loop approximately twice as fast using these twice as large registers, because we have only half the number of instructions.
But now think about what this means. SIMD happens on a single core. We can still create multiple threads and divide up the array to use even more parallelization. On one level, we can compute multiple parts of the array at the same time using multiple threads (that are hopefully scheduled on multiple cores at the same time). But on another level, in each core, multiple operations can be performed at the same time with SIMD. So, SIMD can be combined with multi-threading.

4.3.3 How is Java Code Vectorized?

So, if we have hardware that supports such special SIMD instructions, who actually optimizes the code to behave like this?
Let us quickly recap that in Java, the compiler translates our Java code to bytecode, which is agnostic of the underlying hardware. This means that such optimizations cannot happen at compile-time in Java, because the compiler cannot assume that the underlying hardware actually supports such instructions. But we also remember the JIT compiler that optimizes frequently executed code at *runtime*. Vectorization is exactly such an optimization the JIT compiler can perform. Consider again the code from 4.3. When we look at the machine code the JIT compiler generates at runtime (on an x86 system supporting SIMD instructions), we see the following:

```
vmovdqu   0x10(%r9, %rbx, 8), %ymm0
vmulpd    0x10(%rcx, %rbx, 8), %ymm0, %ymm0
vmovdqu   %ymm0, 0x10(%r13, %rbx, 8)
movslq    %ebx, %rsi
vmovdqu   0x30(%r9, %rsi, 8), %ymm0
vmulpd    0x30(%rcx, %rsi, 8), %ymm0, %ymm0
vmovdqu   %ymm0, 0x30(%r13, %rsi, 8)
vmovdqu   0x50(%r9, %rsi, 8), %ymm0
vmulpd    0x50(%rcx, %rsi, 8), %ymm0, %ymm0
vmovdqu   %ymm0, 0x50(%r13, %rsi, 8)
vmovdqu   0x70(%r9, %rsi, 8), %ymm0
vmulpd    0x70(%rcx, %rsi, 8), %ymm0, %ymm0
vmovdqu   %ymm0, 0x70(%r13, %rsi, 8)
```

This is just a small part of the generated code, but we clearly see two things:

- Vector instructions are used. The instructions starting with 'v' here are part of the AVX instruction set. AVX extends the x86 instruction set with vector (or SIMD) instructions for machines supporting it.
- We see that ymm registers are used. These are special registers that are 256-bits wide, meaning that they can hold *four* doubles. This means (in conjunction with the vector instructions) four times less instructions compared to using normal 64-bit registers with normal (non-vector) instructions.

We can see that the JIT compiler successfully managed to optimize this loop at runtime to use the underlying hardware and vectorize it with SIMD without the programmer having to specify anything. Note however that JIT only optimizes parts of the code that are executed a lot (for example a loop with many iterations). If the loop has only a few iterations in above code, the JIT compiler does not optimize it and hence the code does not use vector instructions even if the hardware supports it.
The performance of Java suffered from this for a long time. Recently, a Vector API was introduced, which provides for example a special array class that is then guaranteed to be mapped to certain vector machine instructions by the compiler. But we will not work with this library in this course.

4.3.4 Concluding Vectorization

What we remember about vectorization is the following:

- Vectorization exploits instruction independence in the special case that the independent instructions are all of the same kind and operate on some vector-like datastructure (for example a for-loop performing an addition on each array element).
- Vectorization needs to be supported by the hardware. Usually in the form of larger registers and multiple execution units (in our example multiple ALUs). Then, there are special instructions that need to be used in order to make use of this.

- The compiler is responsible for translating these vectorizable code snippets into vector instructions for hardware with support for it. But in many languages vectorization is exposed to the programmer via special libraries or the ability to hint to the compiler to use such vector instructions. In Java for example, there is the Vector API that provide guarantees about being mapped to vector instructions (if the underlying hardware supports it).

4.4 Overview

We discussed a few approaches towards improving the performance of a program on a single core. Let us now summarize and compare them to get a feeling for how they differ and what other approaches there are.

- **Vectorization** exploits instruction independence in the special case where the independent instructions are all of the same kind on elements of a vector-like datastructure. The compiler combines multiple instructions (like an addition on multiple array elements) into a single vector instruction, either by finding it itself (auto-vectorization) or by the programmer specifying it in some form.
- **Pipelining** describes that the instruction execution is divided into multiple stages. The execution of independent instructions automatically happens in parallel (in the sense that they overlap in the pipeline). But to maximize the exploitation of ILP, the compiler and hardware need to ensure that dependent instructions have sufficient distance and these gaps get filled with independent instructions.
- **Superscalar execution** means that in each cycle, the CPU does not fetch just one, but *multiple* instructions at once. This means that we need many additional execution units. We can imagine that in this approach, we have multiple mini-cores in one core, which can each fetch an instruction in every cycle and they each have their own execution path. To make use of all this hardware, the thread running on this core needs to have enough ILP, or else many of these mini-cores will have nothing to do. The difference to vectorization is that in superscalar execution, there are actually *multiple* instructions that each have their own execution path on the core. Contrast that to vectorization, where the compiler just combines multiple instructions into a single one operating on larger operands.

4.5 Exercises

1. Pipelining. Over at UZH, the law students have been tasked with writing a legal essay about the philosophy of Swiss law. To write the essay, a student first needs to inform themselves about the subject. To do so, they must read from four different books, each of which contains some information necessary to understand the next book.

Every student takes the same amount of time to read a book, and can only read the books in the specified order:

1) Reading book A takes 80 minutes

2) Reading book B takes 40 minutes

3) Reading book C takes 120 minutes

4) Reading book D takes 40 minutes

Unfortunately, the library only has a single copy of each book.

1. Let's assume all law students are a bit too competitive and don't return any books before they're done reading all of them. How long will it take for 4 students until all of them have started writing their essays?
2. The library introduces a "one book at a time" policy, i.e. the students have to return a book before they can start on the next one. How long will it now take for 4 students until all of them have started writing their essays? What is the throughput of this "pipeline" per hour? What is the latency?
3. Where is the problem with this system? What can the library do to solve this problem?

4.6 Solutions

1. Pipelining.

1. In the case of sequential execution, we simply need to add the execution times of the four stages together and multiply this by the number of students.

$$(80 + 40 + 120 + 40) * 4 = 280 * 4 = 1120 \text{ minutes}$$

2. We assume all students immediately pick up a book as soon as it's available and they've finished reading the previous one. We can then model execution in the following way:

Time (s)	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720
Load 1	A	A	B	C	C	C	D												
Load 2			A	A	B	-	C	C	C	D									
Load 3					A	A	B	-	-	C	C	C	D						
Load 4							A	A	B	-	-	-	C	C	C	D			
Load 5									A	A	B	-	-	-	-	C	C	C	D

We compute the throughput the standard way, i.e. Throughput $= \frac{1}{120}$. Therefore, throughput is 1 student per 2 hours.

When looking at the diagram above we see that the waiting time before students can read book C, and therefore also the latency, increases indefinitely.

3. Latency increasing indefinitely is a sign that the pipeline is unbalanced. There are several solutions to this problem. First, the library could force people to keep each book for exactly 120 minutes or, slightly more efficient, to keep book A for 120 minutes and book B for 80 minutes. Alternatively, the library could buy a second copy of book C. The reader is encouraged to model the execution and convince themselves that this would alleviate the issue. A last option would be to just split book C into two copies, effectively replacing the third stage with two stages, each with an execution time of 60 minutes.

Time (s)	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720
Load 1	A	A	B	C	C‖C'	C'	D												
Load 2			A	A	B	C	C‖C'	C'	D										
Load 3					A	A	B	C	C‖C'	C'	D								
Load 4							A	A	B	C	C‖C'	C'	D						
Load 5									A	A	B	C	C‖C'	C'	D				

5 Measuring Parallelism

In this chapter, we are thinking about parallel programs again. But, instead of concering us with implementing actual programs, we are going to think very abstractly about them. We care about measuring their parallelism and in particular how programs scale with using more processors.

5.1 Performance

5.1.1 Speedup

Before we take a look at different ways of measuring performance, we first need to introduce some terminology. To that end, let P denote the number of processors available during the execution of a program.

Definition 5.1.1. $\mathbf{T_P}$ is the time it takes the program to execute on P processors.

Remark. T_∞ denotes the execution time when we have as many processors at our disposal as is required to get the best-possible execution time.

We will usually use T_1 to talk about the *sequential* and T_∞ to talk about the *minimum* execution time of a program. When talking about performance in general, we will use T_P.

Definition 5.1.2. The **speedup** of a program is

$$S_P := \frac{T_1}{T_P}$$

The speedup is essentially the ratio of the sequential execution time to the execution time given P processors.
Naively, we might expect the speedup of our program to always be $S_P = P$, i.e. $T_P = T_1/P$. This would mean perfect linear speedup. However, in reality we usually see *sub-linear* speedup, caused by additional overheads due to inter-thread dependencies, thread creation, thread communication and memory-hierarchy issues.

5.1.2 Amdahl's Law

Gene Amdahl was an American computer scientist. In 1967, he published a paper titled *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities* that coined *Amdahl's Law*. In the paper, Amdahl was defending single-core processors. As an influential figure, Amdahl's paper had significant impact on the industry and strengthened the position of the single-core processor against upcoming multi-core systems. The argument of Amdahl is simple: Given any program, it has some fraction that is sequential (cannot be parallelized). So, even given a machine with lots of processors, the best that can be achieved is to decrease the time required for the *parallelizable* part of the program. Adding more processors cannot possibly do anything about the purely sequential part of the program. So, even if we have a program that is 90% parallelizable, we cannot get below 10% of its runtime by parallelizing, even with an infinite number of processors. This means that in order to make great improvements in computing speed, the focus should lie on improving the sequential performance of computers instead of focussing on building multi-core systems.
The assumption of Amdahl's law is that the work is fixed. We can visualize the work partitioning on the processors Amdahl's law assumes:

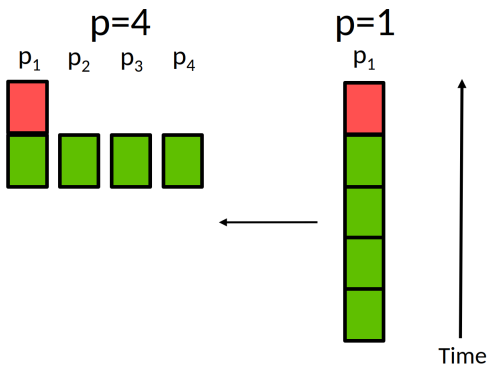


Fig. 22: Red Part is Sequential Work, Green Part is Parallelizable Work.

When we have **P** processors, each processor has to execute $\frac{1}{P}$ of the parallelizable part. One processor additionally has to execute the entire serial part. This one processor becomes the bottleneck and we consider the runtime on **P** processors to be the runtime of this one processor (because the others can execute their part in parallel to it and thus do not prolong the runtime).

Let W_{ser} denote the time spent doing non-parallelizable, serial work and W_{par} denote the time spent doing parallelizable work.
We then write:

$$T_1 = W_{ser} + W_{par}$$

In above figure, W_{ser} is the red part and W_{par} is the four green parts stacked on top of each other. Thus, T_1 is the sum, which amounts to the sequential execution time of the entire progam. Let us now derive Amdahl's law:

W_{ser} remains constant as we increase the number of processors, since the program we execute is fixed (and we assume no multi-processing overhead). Therefore, given P processors, we have the following lower bound on T_P:

$$T_P \geq W_{ser} + \frac{W_{par}}{P}$$

We have an inequality, since *super-linear* is impossible in reasonable theoretical models. Equality would hold when we have perfect linear speedup.
Recall the definition of speedup. Plugging in the relations derived above, we get:

$$S_P = \frac{T_1}{T_P} \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$

Let **f** denote the non-parallelizable, serial fraction of the total work. We then obtain the following equalities:

$$W_{ser} = \mathbf{f} * T_1$$
$$W_{par} = (1 - \mathbf{f}) * T_1$$

This gives us the more common form of Amdahl's Law:

Theorem 5.1. *Let* **f** *denote the non-parallelizable, serial fraction of the total work done in a program and P the number of processors at our disposal. Then, the following inequality holds:*

$$S_P \leq \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}}$$

When we let P go to ∞, we see:

$$S_\infty \leq \frac{1}{\mathbf{f}}$$

In order to see why this is such an important result, we can try plugging in a couple of values. Assume that 25% of a program is non-parallelizable. This means that even with the *IBM Blue Gene/P* supercomputer with its 164'000 cores, we can only achieve a speedup of at most 4. While a depressing result at first sight, this makes perfect sense when we consider the fact that these 25% are completely fixed, in the sense that the execution time can't possibly be reduced past this point.

The conclusion to draw from this result is that when writing parallel programs, the sequential part needs to be reduced as much as possible, for example by only using the `synchronized` keyword (which enforces sequential execution of the corresponding block) where it is absolutely necessary.

5.1.3 Gustafson's Law

After Amdahl's famous paper was published in 1967, the industry remained very skeptical regarding the viability of massively parallel systems. More than 20 years later, times have changed and high-performance computing researcher John Gustafson published a paper titled *Reevaluating Amdahl's Law* in 1988. In this paper, Gustafson deems the assumptions Amdahl made in the formulation of his law as inappropriate for (at this time) new approaches towards massive parallelism. The argument of Gustafson was the following: Amdahl assumed a fixed problem size. In particular did Amdahl assume that W_{par} is independent of **P**, the number of processors. Gustafson argued that this is not realistic, since when a massively parallel system is available for a given task, people are going to take advantage of this. For physical simulations, one can increase the number of timesteps, difference operator complexity and grid

resolution among others. Gustafson argues that one should assume the *run-time* to be constant instead of the problem size. A more modern example is training a large language model. Having massive clusters at their disposal, companies are going to train a model with the highest possible number of parameters in a reasonable amount of time. The limiting factor is the runtime, and the problem size adapts based on how much can be achieved in this time. This assumption of fixed runtime instead of fixed work is the fundamental difference to Amdahl's law. Let us derive Gustafson's law now.

Given **P** processors, Gustafson's law assumes that each processor executes the entire parallel part of the original program and one of these processors additionally executes the sequential part (which we assume to stay fixed when varying **P**). We can visualize this in the following way:

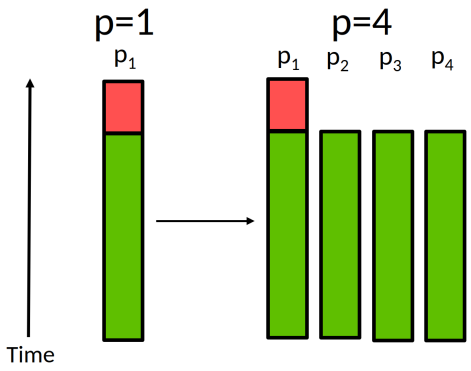


Fig. 23: Red Part is Sequential Work, Green Part is Parallelizable Work.

Note that each processor gets the entire green part to execute. On four processors, more work is done than on one. This was not the case with Amdahl's law.

Let W denote the work done in a fixed time interval. We can write

$$W = \mathbf{f} * W + (1 - \mathbf{f}) * W$$

Where $\mathbf{f}$ is the sequential fraction of T_1. As we increase the number of processors at our disposal, we multiply the parallel fraction of our program. The serial fraction remains the same. Letting W_P be the work done with P processors at our disposal, we get

$$W_P = \mathbf{f} * W + P * (1 - \mathbf{f}) * W$$

In the figure, we can see this as the parallel work (green part) is $(1 - \mathbf{f}) * W$ and each of the **P** processors executes this green part. One processor additionally has to execute the sequential work (red part), which is $\mathbf{f} * W$.
We can now follow *Gustafson's law.*

Theorem 5.2. *Let* $\mathbf{f}$ *denote the non-parallelizable, serial fraction of the total work done in the program and* P *the number of processors at our disposal. Then, we get*

$$\begin{aligned} S_P &= \mathbf{f} + P(1 - f) \\ &= P - \mathbf{f}(P - 1) \end{aligned}$$

We can see this result directly from the figure. Let us normalize the runtime (on **P** processors) to one. We simply ask ourselves how long it takes a single processor to execute all the work that the **P** processors do. And this work is given by the equation $W_P = \mathbf{f} * W + P * (1 - \mathbf{f}) * W$. So, a single processor needs to execute the parallel part **P** times and the sequential part once. We can visualize the speedup now as all the green parts plus one red part stacked on top of each other (runtime on one processor) versus just one red and one green part (runtime on **P** processors, because all the other green parts are computed in parallel and thus do not add to the runtime). If we again consider a program where 25% is non-parallelizable, we get a speedup of 4 already when we increase the number of processors to 5.

It has to be noted that Gustafson's law is only formulated for a certain class of problems where problem size can be scaled when additional computational resources are available. For example Amdahl's law might be better applicable to problems like sorting a fixed set of integers, but Gustafson's law could be used to describe the problem of rendering a 3D-scene as detailed as possible - the level of detail can be increased in the case of 3D-rendering, but there is no such thing as sorting a set of integers in a more exact way.

5.2 Exercises

2. Amdahl's Law, Gustafson's Law, Performance.

1. Suppose a computer program has a method **M** that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on an n-processor multiprocessor machine according to Amdahl's Law?
2. The analysis of a program has shown a speedup of 5 when running on 15 cores. What is the serial fraction according to Amdahl's Law (assuming best possible speedup)?
3. The analysis of a program has shown a speedup of 5 when running on 15 cores. What is the serial fraction according to Gustafson's Law?

5.3 Solutions

2. Amdahl's Law, Gustafson's Law, Performance.

1. We see that the program has a serial fraction of at least 40%. Therefore, we set $\mathbf{f} \geq 0.4$ and insert this into the formula specified by Amdahl's Law:

$$S_P \leq \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}} = \frac{1}{0.4 + \frac{0.6}{P}} \xrightarrow{P \to \infty} \frac{1}{0.4} = 2.5$$

2. Assuming best possible speedup, Amdahl's Law tells us the following holds true

$$S_P = 5 = \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}} = \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{15}}$$
$$\iff$$
$$5\mathbf{f} + \frac{1-\mathbf{f}}{3} = 5\mathbf{f} - \frac{1}{3}\mathbf{f} + \frac{1}{3} = 1$$
$$\iff$$
$$\mathbf{f} = \frac{1}{7}$$

3. Gustafson's Law tells us the following holds true

$$S_P = \mathbf{f} + P(1-\mathbf{f}) = \mathbf{f} + 15(1-\mathbf{f}) = 5$$
$$\iff$$
$$14\mathbf{f} = 10$$
$$\iff$$
$$\mathbf{f} = \frac{5}{7}$$

6 Parallelizing Algorithms

In the previous chapter, we discussed how *Amdahl's Law* tells us how our programs are inherently bottlenecked by their non-parallelizable part, no matter how many processor we have available. Hence, it is our goal to reduce the sequential fraction of our programs when we want to run them on many processors. To this end, we introduce some methods to parallelize algorithms in this chapter, specifically in Java. Here, parallelization means that we divide up the task at hand onto different *threads*, assuming that they are able to run on different *processors*. This is in contrast to Chapter 3, where we assumed a fixed program that will be run on a single processor (imagine this is the work chunk a thread was assigned to process) and analyzed how we can exploit independence within this set of instructions.
We can use the techniques learned in this chapter to design parallel implementations of algorithms.

6.1 Fork-Join Parallelism

Let us once again consider the task of summing up an array. Using task parallelism, we can simply divide up the array on some number of threads. Each thread sums up some part of the array and finally, the main thread sums up these partial results. The corresponding code looks something like this in Java:

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for (int i = 0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start();
    }
    for (int i=0; i < 4; i++) { // combine results
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

Here, `SumThread` is some `Thread` class summing up the given indices. We do not really care about its exact implementation here.

This approach of dividing up the work into multiple different tasks and then combining their results is called *Fork-Join Parallelism*. The join in fork/join does not necessarily refer to the `Thread.join()` method, where we wait for a thread to finish, but can also mean the combination of the partial results (in this case the main threads sums up the partial results).

6.1.1 Parallel Divide-And-Conquer

Above solution is not new to us. But we will improve upon it. Assume we want to run above code on some hardware efficiently. We find a few things that are not optimal with this general approach:

- There is a fixed number of threads created (four in this case), no matter how many processors the underlying hardware has. We can solve this by making the number of threads created a parameter of the method.
- For general datastructures, dividing it into equal chunks for threads to process can lead to large workload imbalances across threads. Consider a graph datastructure. When we give the same number of vertices to each thread to process, the number of edges in a vertex set might be vastly different. When we now want to run parallel BFS, we will not get good speedup, because the few threads with the most edges will bottleneck the runtime. In this case, this is not a problem, since each thread will have about the same workload with summing array entries.
- We have a sequential bottleneck of summing up the partial results in the end.

We can improve on the last point by parallelizing the result combination. This can easily be implemented by recursively dividing the tasks with the Fork-Join model. This gives us a parallel version of the *divide-and-conquer* paradigm. The basic structure of a divide-and-conquer program is the following:

```
Divide and Conquer:
  if cannot divide:
    return unitary solution (stop recursion)
  divide problem in two
  solve first (recursively)
  solve second (recursively)
  combine solutions
  return result
```

Fig. 24: Structure of a Divide-and-Conquer Program.

We can implement this with the following code:

```java
public static int do_sum_rec(int[] arr, int startIdx, int endIdx) {
    int size = endIdx-startIdx;
    if (size == 1) // check for termination criteria
        return arr[startIdx];
    // split array in half and call self recursively
    int mid = size / 2;
    int sum1 = do_sum_rec(arr, startIdx, startIdx + mid);
    int sum2 = do_sum_rec(arr, startIdx + mid, endIdx);
    return sum1 + sum2;
}
```

We can trivially parallelize this by solving the recursion for the first and second subproblem in parallel. We get the following structure of function calls:

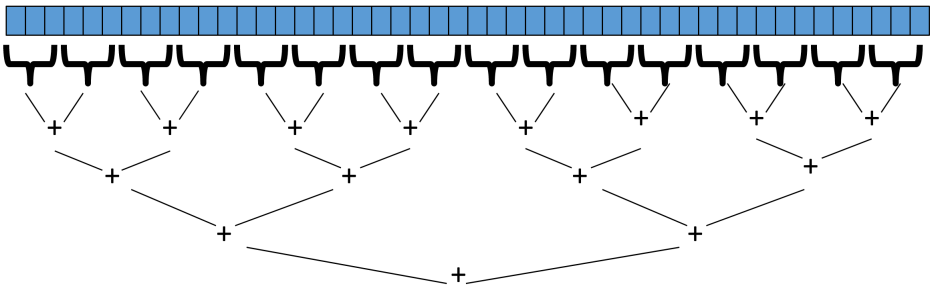


Fig. 25: Structure of a Divide-and-Conquer Program.

Function calls on the same level in the tree can be executed in parallel. Notice that the tree has a depth logarithmic in the number of array entries. This is much better than what we can achieve by simply dividing up the array between threads.
Although to actually get the logarithmic runtime, we require as many processors as array entries. Else, the threads on the last level cannot all run in parallel. But this approach is still superior compared to our initial approach with dividing the array into equal chunks, because it solves two previously mentioned problems:

- The sequential bottleneck of summing up the partial results is resolved since even the result combination is parallelized, allowing for more speedup with increasing number of processors.
- It is easier to implement more equal load balancing. Consider again a graph as an input datastructure. When a task divides up its assigned vertices on two subtasks, it can also count the number of edges in the two sets and adjust the division accordingly (such that both subtasks get not just a similar amount of vertices, but also a similar amount of edges). This is a much simpler problem than when we have to partition the entire graph on n threads in the beginning.

6.1.2 Parallel Divide-And-Conquer with Java Threads

But now, how do we implement this parallelization of the two subtasks? Of course we can create two threads solving each half recursively. This would look something like this:

```
public void run() {
    int size = endIdx - startIdx;
    if (size == 1) {
        result = arr[startIdx];
        return;
    }
    int mid = size / 2;
    SumThread t1 = new SumThread(arr, startIdx, startIdx + mid);
    SumThread t2 = new SumThread(arr, startIdx + mid, endIdx);
    t1.start();
    t2.start();
    t1.join(); // join() would have to be in a try-catch block
    t2.join();
    result = t1.result + t2.result;
    return;
}
```

Now imagine we want to process arbitrarily large arrays like this. Remember that each Java thread is mapped to an OS thread. Also remember that each OS thread gets some resources, for example a stack. This is a small region of memory residing in main memory. When we now create millions of threads for a large array, we will sooner or later run out of memory. This will be visible with the following error:

```
[4.570s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes:
stacksize: 1024k, guardsize: 0k, detached.
[4.570s][warning][os,thread] Failed to start the native thread for java.lang.Thread "Thread-20000"
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or proce
ss/resource limits reached
        at java.base/java.lang.Thread.start0(Native Method)
        at java.base/java.lang.Thread.start(Thread.java:1535)
        at out.mem(outOfMem.java:23)
        at out.main(outOfMem.java:6)
```

Fig. 26: OutOfMemoryError-Exception due to creating too many threads in Java.

Implementing parallel divide-and-conquer with native Java threads is not feasible due to their resource consumption. Using native Java threads for recursive subtasks is not suitable for a number of reasons anyways:

- **Resource blocking:** A thread will join the new threads it creates and potentially wait for a long time for them to return. In this time, the Java Thread object blocks an OS thread, which may be idle (does not use up CPU time, which is good), but still blocks the resources allocated for it. When we have millions of threads, most of which are currently joining, large chunks of memory are allocated, even though few threads are doing actual work.
- **Many more threads than cores:** We probably do not have millions of cores to run the code on. Therefore, we cannot possibly gain an advantage by creating so many (Java and thus OS) threads. Remember that creating OS threads brings some overhead. Also, context switching between the threads is expensive, so we never want to have many more active OS threads than we have processors.

The problem we have is fundamentally that Java threads are too *heavy-weight* for these recursive tasks. The recursive tasks do not each require their own OS thread, which is what is given to them when we create Java threads.
Remember that in chapter 1, we said that many languages have something like virtual threads. That is, the user creates virtual threads (or tasks) and the language runtime decides itself how these are going to be mapped to OS threads. The advantage is that by using this approach, the virtual threads are (potentially) much less heavy-weight than when simply one-to-one mapping to OS threads, solving all of our problems.

6.1.3 Manually Optimizing Divide-And-Conquer using Java Threads

Before we look into solutions to this problem of Java threads being too heavy-weight, let us perform two manual optimizations:

1. Currently, we break down the problem until the assigned chunk size equals one. This is not efficient, as thread creation brings some overhead. Say we process a chunk of 20 array elements. When we divide this onto two new threads, each only has to process 10 elements. However, the time required to create and start new threads is significantly longer than the time gained to process 10 instead of 20 elements. Hence, we introduce a **cutoff** at, say, 1000 elements. As soon as the assigned array chunk has length less than 1000, the thread will sum it itself and not divide it any further. Using this optimization, we need to create fewer threads.

2. When we create two threads to solve each half of the problem, the thread creating them has nothing to do while waiting on the two results but still blocks resources. It would be more efficient when the thread solves one half itself and outsources the second half to another thread to solve concurrently. We can do this by calling `Thread.run()` instead of `Thread.start()`. This again means that we need to create fewer threads.

Using these two optimizations, we receive the following code:

```
public void run() {
    int size = endIdx - startIdx;
    if (size <= 1000) {
        result = 0;
        for (int i = startIdx; i < endIdx; i++) {
            result += arr[i];
        }
        return;
    }
    int mid = size / 2;
    SumThread t1 = new SumThread(arr, startIdx, startIdx + mid);
    SumThread t2 = new SumThread(arr, startIdx + mid, endIdx);
    t1.start();
    t2.run(); // executes the run() method of t2 with the current thread
    t1.join(); // join() would have to be in a try-catch block
    result = t1.result + t2.result;
    return;
}
```

Even with these optimizations, our problem with Java threads being too heavy-weight remains due to resource-blocking, creation and context switching overhead. These optimizations simply mean that we create less threads overall, but for sufficiently large arrays, the exact same problems occur.

6.1.4 Solving Heavy-Weight Threads in Java: ExecutorService

Java provides us the `ExecutorService` framework as an abstraction layer between tasks and threads. Instead of creating a thread to solve one half of the array recursively, we create a *task* and submit it to an `ExecutorService`. The `ExecutorService` then maps these tasks to Java threads itself. This scheduling of tasks to Java threads is similar to what the OS does on a lower abstraction level, where it maps OS threads to available processors.

This seems to be exactly what we need to solve our issue with the heavy-weight Java threads. Let us create an `ExecutorService` instance in Java:

```
// Create an ExecutorService with a fixed thread pool of 4 threads
ExecutorService executor = Executors.newFixedThreadPool(4);
```

Here, we give the `ExecutorService` instance a pool of four threads. This means that the service will create four threads when initialized. When we now submit tasks, these will wait in a queue for a thread in the pool to become available.

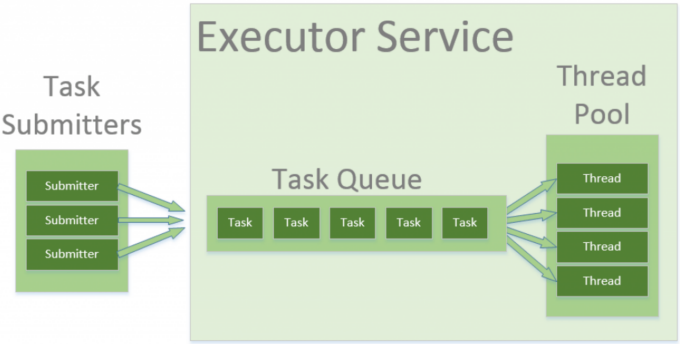


Fig. 27: Internal structure of `ExecutorService` with a fixed thread pool: https://www.baeldung.com/thread-pool-java-and-guava

What exactly do we submit to the `ExecutorService` now? Threads? Almost. Remember that we could create a Java Thread object by initializing it with a `Runnable` object? We can submit exactly such `Runnable` objects to the `ExecutorService` (and also others). Some important methods of `ExecutorService` are the following. Let *ex* be an `ExecutorService` instance in our code.

- `ex.submit(Runnable task)`: Submits a Runnable object for execution and returns a Future object representing that task.
- `ex.submit(Callable<T> task)`: Submits a value-returning task for execution and returns a Future object representing the pending results of the task. This allows us to submit a `Callable<T>` object (think of a Runnable object that can return a value of type T and implements the `call()` function instead of the `run()` function).
- `ex.shutdown()`: Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. After all previously submitted tasks are executed, the thread pool is deallocated. We always call this method when we finish using the `ExecutorService` instance.

When we submit tasks, `ExecutorService` returns us a Future object. This provides us a handle for the submitted task to for example request the result or check its progress. Let us write a simple program using `ExecutorService` to get a better idea of what we can do.

```
public static void main(String[] args) {
    /* submit 100 callables to the service and store the returned
     Future objects in a list */
    ExecutorService ex = Executors.newFixedThreadPool(4);
    List<Future<Integer>> futures = new ArrayList<>();
    for (int i = 0; i < 100; i++) {
        Future<Integer> future = ex.submit(new MyCallable());
        futures.add(future);
    }

    // sum up the results of all tasks
    int sum = 0;
    for (Future<Integer> future : futures) {
        try {
            sum += future.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }

    System.out.println("Sum of all results: " + sum);
    ex.shutdown();
}
```

```
class MyCallable implements Callable<Integer> {
    /* implement the call() method. The call() method is the code the task
     executes when it is its turn */
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            sum += i;
        }
        return sum;
    }
}
```

We can see that when calling `future.get()`, we get the result of the computation. When using threads, we first need to call `thread.join()` to ensure the computation is completed. But here, the underlying threads are abstracted away and calling `future.get()` automatically waits on the task being finished, without having to join.

6.1.4.1 Parallel Divide-And-Conquer with `ExecutorService`

With `ExecutorService`, we can now try implementing our parallel divide-and-conquer program again. Since we can create a fixed thread pool, we will not have the same issue of running out of memory, no matter the array size. Let us implement the `call()` method of a corresponding `Callable` class using `ExecutorService`:

```
public class SumRecCall implements Callable {
    ...

    public Integer call() throws Exception {
        int size = endIdx - startIdx;
        if (size <= 1000) {
            int result = 0;
            for (int i = startIdx; i < endIdx; i++) {
                result += arr[i];
            }
            return result;
        }
        int mid = size / 2;
        SumRecCall c1 = new SumRecCall(ex, arr, startIdx, startIdx + mid);
        SumRecCall c2 = new SumRecCall(ex, arr, startIdx + mid, endIdx);
        Future<Integer> f1 = ex.submit(c1);
        Future<Integer> f2 = ex.submit(c2);
        return f1.get() + f2.get();
    }

    ...

}

// main method defined somewhere else
public static void main(String[] args) {
    ExecutorService ex = Executors.newFixedThreadPool(4);
    SumRecCall task = new SumRecCall(ex, arr, 0, arr.length);
    Future<Integer> result = ex.submit(task);
    System.out.println(result.get());
    ex.shutdown();
}
```

When we now run this program, we notice that it does not return. To understand what happens, let us consider the workings of `ExecutorService` again. A submitted task is put in a wait queue, where it waits until one of the four threads in the pool is free. Now, let us consider what happens with the first five tasks submitted in this code:

- The first task gets the whole array to process, hence it creates and submits two further tasks to process each half. However, this first task is assigned to the first thread in the pool. It will keep occupying this thread, even when it now has to wait for its subtasks to complete.
- The two tasks created by the first task occupy the second and third thread in the pool, meaning only one thread remains free. Since the array is presumably quite long, these two tasks will again create two more tasks each (to solve each of their problem half) and submit them to the service. Again, even though they are waiting for a Future to return its result, they all occupy a thread in the pool and will not let it go until they completed their `call()` method.
- Task four will get the last thread in the pool. The fifth task that is submitted will not find a free thread in the pool anymore. That means that it will wait in the queue for a thread to become free. However, this will not happen, as the tasks currently occupying the thread are waiting for the tasks in the queue to return their result. They of course cannot do this without a thread. We have reached a deadlock.

This seems very disappointing at first. The fixed thread pool of the executor service is not made for tasks that create further tasks themselves. An example for a useful application using the fixed thread pool of `ExecutorService` is handling HTTP requests of a webserver. Each request is submitted to the `ExecutorService` as a task and the service distributes the tasks across the available threads.

6.1.5 Java Fork/Join Framework

When we browse the Java documentation, we find that `ExecutorService` is an interface. Remember that interfaces in Java define some high-level behaviour by declaring methods that classes can then specifically implement. We can instantiate an `ExecutorService` for example through the fixed thread pool we saw earlier. There is another interesting implementation of the `ExecutorService` interface called `ForkJoinPool`. This class is the center of Java's fork/join framework and designed for recursive tasks, exactly what we are looking for. We can instantiate a `ForkJoinPool` in the following way:

```
ForkJoinPool fj = new ForkJoinPool(4);
fj.invoke(new Task());
fj.shutdown();
```

Assuming that the class `Task` is some kind of generic task the `ForkJoinPool` accepts, this code creates a `ForkJoinPool` that uses a maximum of four Java threads and starts a `Task` object. With our previous knowledge, we can characterize the `ForkJoinPool` class with a few points:

- A `ForkJoinPool` is an implementation of the `ExecutorService` interface. We can instantiate it (among other possibilities) by specifying the maximum number of used Java threads as an argument. When we do not give any argument, the thread pool will by default contain as many threads as there are cores available on the system.
- The difference to the `ExecutorService` thread pools we previously saw is that it employs **work stealing**. We will elaborate on this, but in a nutshell, it means that instead of simply waiting for subtasks to finish, threads in the fork/join Framework steal tasks from other threads to execute in the meantime. This is the key feature that prevents the deadlock problem we previously had with `ExecutorService`.
- The tasks that a `ForkJoinPool` accepts are `ForkJoinTasks`. The `ForkJoinTask` class is an abstract base class, with the relevant subclasses being `RecursiveTask<V>` and `RecursiveAction` (similar to `Callable<T>` and `Runnable`). So, we invoke our `ForkJoinPool` usually with either a `RecursiveTask<V>` (when we require our tasks to return a value of type V) or a `RecursiveAction` object (if no return value is required).
- To start subtasks from within a task, we can call `ForkJoinTask.fork()` and then `ForkJoinTask.join()` to wait for the computation to be finished.

6.1.5.1 Parallel Divide-And-Conquer with Fork/Join Framework

Let us now finally implement our parallel array sum algorithm using the fork/join framework:

```
public int sumArray(int[] arr) {
    ForkJoinPool fj = new ForkJoinPool(4);
    SumRecCall sumTask = new SumRecCall(arr, 0, arr.length);
```

```
    int result = fj.invoke(sumTask);
    fj.shutdown();
    return result;
}

class SumRecCall extends RecursiveTask<Integer> {
    int[] arr;
    int endIdx, startIdx;

    // constructor
    public SumRecCall(int[] arr, int startIdx, int endIdx) {
        this.arr = arr;
        this.startIdx = startIdx;
        this.endIdx = endIdx;
    }

    public Integer compute() {
        int size = endIdx - startIdx;
        if (size <= 1000) {
            int result = 0;
            for (int i = startIdx; i < endIdx; i++) {
                result += arr[i];
            }
            return result;
        }
        int mid = size / 2;
        SumRecCall first = new SumRecCall(arr, startIdx, startIdx + mid);
        SumRecCall second = new SumRecCall(arr, startIdx + mid, endIdx);
        first.fork();
        second.fork();
        int firstSum = first.join();
        int secondSum = second.join();
        return firstSum + secondSum;
    }
}
```

We see that the syntax is similar to our previous approach using a fixed thread pool for an `ExecutorService`. Differently though, we do not need to store a `Future` object returned by the task. This is because the `RecursiveTask` *is* the `Future` object (the `RecursiveTask` and `RecursiveAction` classes implement the `Future` interface). Hence, we can simply get the result by calling `ForkJoinTask.join()`.

Note that we could compute one half on the current thread by calling `second.compute()` instead of `second.fork()` to perform the same manual optimization as we did when using Java threads. However, this provides us no advantage in the fork/join framework as the task scheduler of the framework optimizes this by itself. So, we can neatly `fork()` and `join()` both tasks and need not worry about performance. However, the optimization to set a sequential cutoff is still important to make sure each task provides enough work. This is because while we do not create a new thread for each task anymore, there is still some overhead to enqueue this task and schedule it on the worker threads in the pool. Setting the threshold to something like 1000 array elements is reasonable in this case to strike a good balance between exploiting parallelism and ensuring the task creation overhead does not dominate execution time.

6.1.5.2 Work-Stealing in the Fork/Join Framework

We mentioned that the fork/join framework solves the deadlock problem other `ExecutorService` implementations have by means of employing a work-stealing algorithm for scheduling submitted tasks among threads in the pool.

Remember from 6.1.4 (`ExecutorService` introduction) that the submitted tasks within an `ExecutorService` enter a queue where they wait for a free thread in the pool. The fork/join framework now implements work-stealing in the following way:

- Each thread in the `ForkJoinPool` additionally maintains its own double-ended queue of assigned tasks.

- When a thread in the pool runs out of tasks (i.e. its queue becomes empty), it attempts to *steal* tasks from the queues of other threads in the pool.
- When a thread in the pool encounters a `ForkJoinTask.join()` operation (meaning it has to wait for the result/termination of the computation) it processes other tasks from the queue until the joined task is finished.

With this additonal queue per thread and the threads processing other tasks upon a `join()` operation, the recursion required by parallel divide-and-conquer is finally possible in an efficient manner. We can visualize the workings of the fork/join framework in the following way:

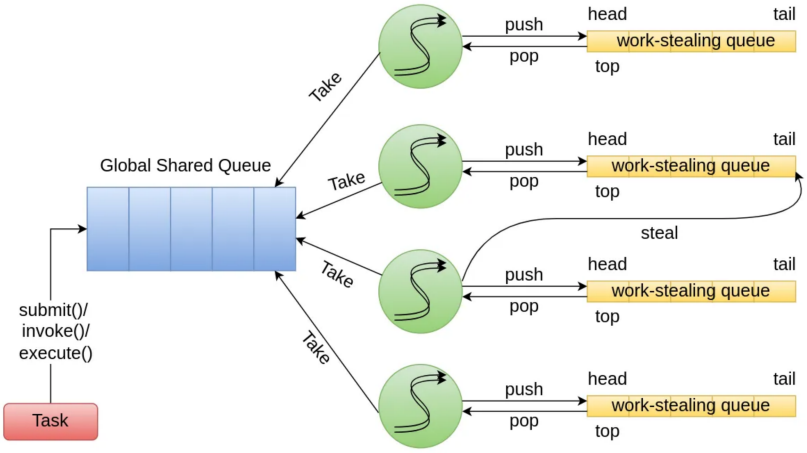


Fig. 28: Task distribution in a `ForkJoinPool` with four worker threads: https://krishnakishorev.medium.com/java-multithreading-concurrency-and-parallelism-part-22-3-9bc0429adc2b

Tasks submitted to the global queue can be taken by worker threads of the `ForkJoinPool`. Tasks that are `fork()`ed are directly pushed onto the executing thread's own queue (and not put into the global queue first). We can also see that threads steal tasks from the *tail* of other queues, while they *push* and *pop* tasks to and from the *head* of their own queue. This is an implementation detail to minimize collisions.

6.1.6 Cilk

Cilk is a programming language developed at MIT in the 1990s. Based on the C programming language, the idea behind Cilk was to extend C to better support multi-threaded programming. You might ask now why we even talk about Cilk in this course, as we already use Java. We are not concerned with actually learning the Cilk language or understanding its exact implementation details.
Cilk is relevant for us as it pioneered a specific programming style of spawning and joining tasks (i), using a work-stealing scheduler to map these tasks to available processors (ii), providing strong runtime guarantees based on this scheduling (iii) and introducing a multi-threaded computation model (iv). If you noticed that this sounds a lot like the Java fork/join framework, you are right. The fork/join framework in Java is based heavily on the Cilk language and thus we can learn a lot about it by studying the ideas behind Cilk.
The basis of Cilk-style programming is that the programmer expresses parallelism by spawning tasks and then joining them (waiting for results). A core of the project was to support fork/join parallelism by implementing a suitable task scheduler. The idea being that the programmer is concerned with the structure of the program (which tasks to spawn and join) and the scheduler is concerned with efficiently mapping these spawned tasks to available processors.

6.1.6.1 The Cilk Model of Multi-Threaded Computation

Closely tied to the Cilk programming style with spawning and joining tasks, Cilk also introduced a graph computation model that allows us to reason about runtime bounds. To introduce the model, we consider the following pseudo-code of the exponential Fibonacci algorithm:

```
public long fib(int n) {
    if (n < 2)
        return n;
    spawn task for fib(n-1);
    spawn task for fib(n-2);
    wait for tasks to complete
    return addition of results
}
```

In the Java fork/join framework, the two spawns would correspond to `ForkJoinTask.fork()` operations and the wait to `ForkJoinTask.join()` operations.
We now consider the computation of `fib(4)`. We can model the program execution as a dag (directed acyclig graph) by:

- Creating vertices for each `spawn` and `wait`.
- Grouping vertices within the same function call (for example within a call to `fib(3)`) into a **procedure** block.
- Creating a **spawn edge** for each spawn call, a **return edge** for each wait and a **continuation edge** for a step within the same function call (procedure).

We receive the following execution graph:

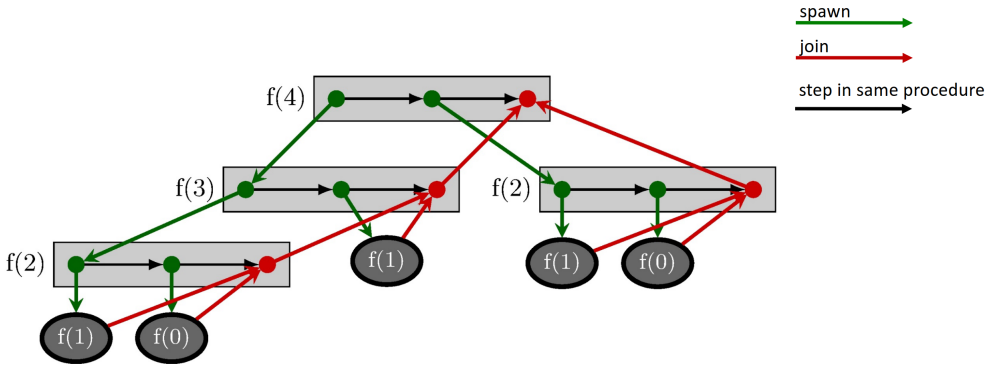


Fig. 29: Cilk Graph of Fib(4) execution.

In each procedure (`Fib(n)` call), we have three vertices corresponding to two spawns and one wait. Exceptions are the leaves, which correspond to the base cases ($n < 2$) and thus immediately return without any vertices.

6.1.6.2 Task Graphs

The Cilk model of multi-threaded computation is an example of a task graph model. Such models help us visualize and understand dependencies in the code to see where we can exploit parallelism. We can simplify the Cilk model to model each procedure as a vertex and only model the spawn edges. The `Fib(4)` execution would then look like this:

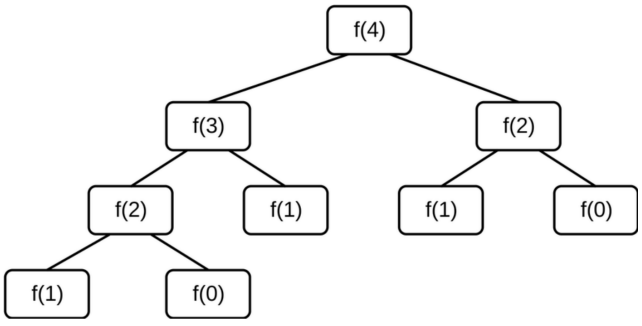


Fig. 30: Simple Task Graph of Fib(4) execution.

With such a simple model, we sacrifice some precision in the dependencies, but we can reason about performance bounds in a more straight-forward manner.
Let us define that each vertex (task) takes one unit of time to execute. Then, $T_1 = n$, where n is the number of vertices in the task graph. This is because a single processor needs to sequentially execute one vertex after the other.
We can see that this kind of task graph is a dag. Branches of this dag have no dependencies between each other and could thus be executed in parallel. This also means that intuitively, the *wider* the task graph, the more parallelism can be exploited.

We notice that no matter how many processors we have, we are limited by the longest path in the graph. This is because each vertex in the path depends on its predecessor. Thus, each path (and in particular the longest path) has to be executed sequentially. Based on this observation, we can define the following:

Definition 6.1.1. The **Critical Path** of a task graph is its longest path. The **Span** $= T_\infty$ is the length of the critical path, that is, the time required to execute all vertices along the critical path.

In our simple model where each vertex takes exactly one unit of time to execute, the span is thus simply the number of vertices lying on the longest path. In the case of our `Fib(4)` task graph, the span is four. With the span, we can define *parallelism* within a task graph or program:

Definition 6.1.2. Parallelism is the maximum possible speedup $\frac{T_1}{T_\infty}$.

This definition of parallelism formalizes our intuition that a wider task graph equates better parallelism, since a wider graph means more total vertices (i.e. T_1 increases), but generally not more depth (i.e. T_∞ stays constant).

With the definition of span we can also formulate the *span law*:

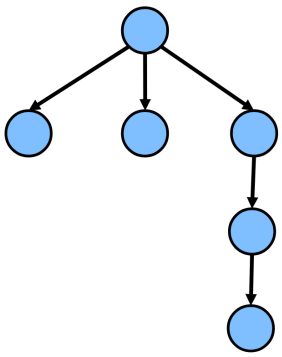
Theorem 6.1. ***Span law****:* $T_P \geq T_\infty$
That is, any execution using P processors is lower bounded by the span of the task graph.

The law follows trivially from the fact that the span is equal to the execution time on an infinite number of processors T_∞ and thus any execution on any finite number of processors P cannot be faster than this.

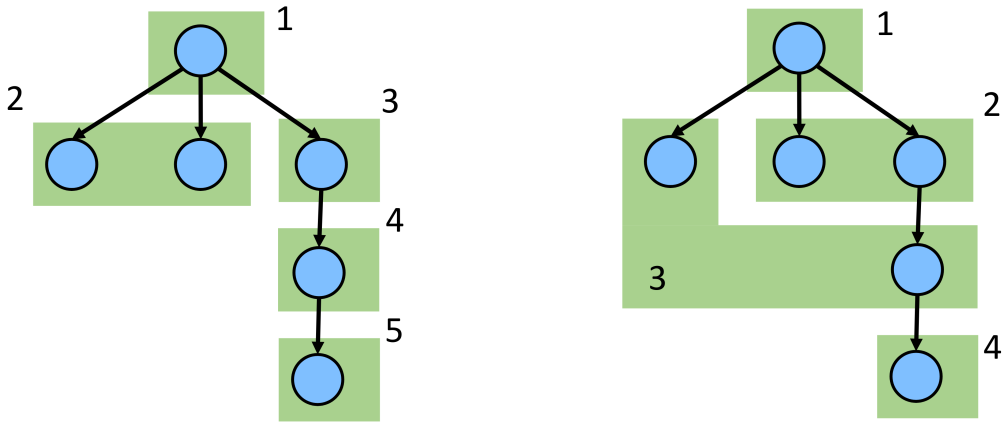
We can also formulate the *work law*:

Theorem 6.2. ***Work law****:* $T_P \geq \frac{T_1}{P}$
That is, P processors require at least $\frac{T_1}{P}$ *time to execute all vertices in the task graph.*

This is because in each unit of time, each processors can execute at most one vertex (due to dependencies, it might be less than one per processors).
We can only give bounds on T_p because it depends on how we schedule the vertices (tasks) on the available number of processors. Let us illustrate this on the following task graph:



What is T_2 here? Using our previously deduced bounds, the *work law* tells us that $T_2 \geq \frac{6}{2} = 3$ since $T_1 = 6$ and the *span law* tells us that $T_2 \geq 4$, since the critical path has length four. However, the exact value of T_2 depends on the scheduling we choose. Consider the following two possible schedulings:



In the first case, we can only exploit the second processor in the second time step. With this scheduling, $T_2 = 5$. However, with the second scheduling, $T_2 = 4$, which is optimal according to the span law.

6.1.6.3 Runtime Guarantees of Cilk

To get back to the subject matter, why did we introduce task graphs in this part of the course? Cilk introduced its task graph model in order to be able to prove theorems and thus provide guarantees about the execution of its programs.
In the previous section, we learned that T_P depends on the scheduling. A core of the Cilk language is its work-stealing scheduler that can provide strong runtime guarantees that are provable within this task graph model. In particular, Cilk gives us the following guarantee about the execution time on P processors:

$$\mathbb{E}(T_P) = \mathcal{O}\left(\frac{T_1}{P} + T_\infty\right)$$

That is, the expected time required to execute a Cilk program on P processors is equal to the critical path length T_∞ plus the sequential time T_1 divided by the number of processors P. This is a strong guarantee, as both T_∞ and $\frac{T_1}{P}$ are lower bounds on any parallel execution (span law and work law).
This means that a Cilk program has an expected runtime that is within a constant factor of optimal. The guarantee can only be given with regards to the expected value (and not about worst-case), because the work-stealing algorithm employs some randomization (for example randomizing which thread to steal work from). Of course we know that the O-notation can hide enormous constants. However, empirical testing showed that the runtime of Cilk programs is very close to optimal with small constants, that is, $T_p \approx \frac{T_1}{P} + T_\infty$.
The important part for us in this course is that the Java fork/join framework is based on Cilk, adopting both its spawning/joining programming style and its work-stealing scheduler, thus also inheriting this strong runtime guarantee.

6.1.7 Summary: Fork/Join Programming in Java

After an odyssey where we...

1. Recognized the **value of parallel divide-and-conquer** due to the simple program structure, ability to easily implement effective load balancing and ability to parallelize even the result combination.

2. Recognized that using **Java threads for fork/join is not optimal**, since they block resources (memory, OS threads) and bring a lot of overhead (bookkeeping, context switching). To solve all these problems, we stumbled upon the `ExecutorService` framework, that offers light-weight tasks instead of heavy-weight threads. We had to notice that the fixed thread pool of the `ExecutorService` is not designed for recursion.

3. Found that the **Java fork/join framework** is exactly what we were looking for and enables us to reap all the benefits we originally hoped for from implementing parallel divide-and-conquer.

4. Learned how the **Cilk programming language** pioneered multi-threading in programming languages and contributed a work-stealing scheduler enabling efficient fork/join parallelism that inspired Java to adopt most of its features within the fork/join framework. We learned that by modelling programs with task graphs, Cilk manages to prove asymptotic lower bounds on execution time that thus also apply to the Java fork/join framework.

... we might feel a bit lost about where we end up now. We have not yet actually learned how to systematically parallelize algorithms, but we have acquired a critical tool towards implementing them efficiently in general (with the programming style of forking and joining tasks) and in particular using the fork/join framework in Java. In the case of the fork/join framework, we even have the guarantee that our programs will run in asymptotically optimal expected time (in the number of processors P) $\mathbb{E}(T_P) = \mathcal{O}\left(\frac{T_1}{P} + T_\infty\right)$ which gives us a solid foundation to write parallel algorithms on.
We also learned how to model parallel programs using task graphs. This helps us think about the parallelism of programs more easily and enables the distinction between work (T_1) and span (T_∞). Our goal for parallel algorithms in terms of performance is thus to reduce span in particular.

6.2 Parallel Patterns

Now we can finally talk about parallelizing algorithms. To be able to do so systematically, we need to look out for patterns in our code. Some important patterns are introduced in this section.

6.2.1 Maps and Reductions

We start with the two most straightforward and also most common patterns and add complexity later.

6.2.1.1 Reductions

Remember all the work we had to do in order to parallelize array summation using the fork/join framework in Java? The good news is that many problems can be effectively parallelized almost exactly like our array summation. Here are a few problems that can be efficiently parallelized using an approach almost exactly to our divide-and-conquer array summation:

- Return the maximum or minimum element of an array.
- Return the number of elements in an array that fulfill some property.
- Return the leftmost element of an array fulfilling some property.

The only parts about the fork/join algorithm that are subject to change are the base case and the combination of results. So, say we want to return the number of elements fulfilling some property:

- For the base case, return 1 if array[startIdx] fulfills the property and 0 otherwise. If we implement a cutoff, which is more efficient, simply loop over the assigned chunk (of about 1000 elements) and count how many elements fulfill the property.
- To combine the two joined subtasks, simply add the results.

Problems of this form are called **reductions**. A reduction is an operation that produces a *single* answer from a collection (for example an array or linked list) via an *associative* operator. This single answer can for example be a count, an element of the collection itself, a boolean value or even something like a histogram. The associativity of the operation is important such that even when dividing and recombining partial results, the end result does not change. If the operation was not associative, we could not use a divide-and-conquer strategy.

6.2.1.2 Maps

A **map** is an operation characterized by two properties:

1. A map operates on each element of a collection (for example an array or a linked list) independently.
2. A map outputs a new collection of the same size.

An operation that fulfills this is for instance vector addition. We start with a vector X and add a vector Y (of the same size) to it. This operates on each element of X independently and results in a new vector Z of the same size as X. Note that we do not perform the *same* operation on each element of X (to each X[i], we add Y[i], which is presumably different for each i), but this is not a requirement of a map operation.
Maps are often called *embarrasingly parallel*, since each element can be processed independently by definition and thus parallelizing a map is trivial.
We can implement maps using the fork/join framework as well. Consider again vector addition. To plan out how we use the fork/join framework to parallelize this, we consider the following:

1. For the `ForkJoinTask` class, we choose to subclass `RecursiveAction` (instead of `RecursiveTask<T>`) since working with an array reference is more efficient than actually returning an array copy. So, we do not need a return value.
2. For the `RecursiveAction` class, we need to store the two input arrays, a result array, a start index and an end index.
3. The base case is simply a for loop to sequentially sum the assigned array chunk (defined by start index and end index).
4. For the recursion, we simply need to fork two tasks to sum each half of the assigned chunk. We do not need to combine these result in any way.

Based on this, we write the following `RecursiveAction` subclass:

```
class VecAdd extends RecursiveAction {
    private final int SEQUENTIAL_CUTOFF = 1000;
    private int startIdx, endIdx;
    private double[] res, arr1, arr2;

    // constructor
    VecAdd(int startIdx, int endIdx, double[] res, double[] arr1, double[] arr2) {
        this.startIdx = startIdx;
        this.endIdx = endIdx;
        this.res = res;
        this.arr1 = arr1;
        this.arr2 = arr2;
    }

    protected void compute() {
        if (endIdx - startIdx <= SEQUENTIAL_CUTOFF) {
            for (int i = startIdx; i < endIdx; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (endIdx + startIdx) / 2;
            VecAdd left = new VecAdd(startIdx, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, endIdx, res, arr1, arr2);
            left.fork();
            right.fork();
            left.join();
            right.join();
        }
    }
}
```

To perform vector additions now, we can write the following method where we initialize a `ForkJoinPool` and start the initial task:

```
public double[] vectorAdd(double[] arr1, double[] arr2) {
    double[] res = new double[arr1.length]; // initialize empty result array
    ForkJoinPool fj = new ForkJoinPool();
    RecursiveAction vec = new VecAdd(0, arr1.length, res, arr1, arr2);
    fj.invoke(vec);
    fj.shutdown();
    return res;
}
```

This is just a reminder on how to use the fork/join framework. We see that implementing maps is very straightforward and even simpler than implementing reductions, because we do not need to combine the partial results of the subtasks.

6.2.1.3 Digression: Map/Reduce vs Fork/Join in Distributed Systems

We may now think that we have a clear hierarchy, where maps and reductions are high-level patterns that can be exploited by means of fork/join-style programming. However, there is a paradigm called map/reduce that is based on maps and reductions, although it views maps and reductions a bit differently. Map/reduce is the basis of computing in distributed systems, while fork/join is utilized mostly on shared memory systems. Let us elaborate on the difference between these two terms:

In this course, we focus almost solely on parallel programming within a single computer in a shared memory environment. This means that the different entities partaking in the parallel computing (the processors/cores) share memory and can thus efficiently communicate with each other. There is also *distributed computing* operating on a larger scale with massive datasets, where the entities are called nodes and are usually whole computers. There is no shared memory between the nodes (communication happens by sending messages and is expensive) and data is distributed as well (each node has some part of the data while in shared memory computing, all processors can access the entire data). Hence the paradigms used to parallelize computations are different. While in shared memory computing, fork/join parallelism is popular, map/reduce is the workhorse of distributed computing.

To understand why fork/join does not make so much sense in distributed computing, let us introduce an analogy. We can think of the distributed computing environment as the Holy Roman Empire, where we want to conduct a concensus. The data we operate on are the citizens and each city can be considered a node. The data is not shared, as each city can only count its own inhabitants. So, we would let each city locally count its inhabitants (all cities in parallel) and then let them send their results back to the capital, where the partial results are summed (or *reduced*) to the final answer. Generally, everything that can be computed per node is considered a map in distributed map/reduce (even though not strictly a map according to our definition) and only when we have to combine the partial results of each node we call it a reduction.

A divide-and-conquer approach would not make much sense here. Cities would have to exchange their results with each other and sum them, but considerable distances would have to be covered to communicate these results between cities. It requires much less communication and is more efficient to just let each city send a representative to the capital once. Note however that each city can be considered a shared memory environment itself and can parallelize its local counting (where communication is cheaper) using a divide-and-conquer approach.

This example is not a perfect analogy, but it gives us sufficient intuition on why fork/join and map/reduce coexist as paradigms - they are designed to exploit parallelism in different environments and on a different scale.

To conclude, map/reduce makes sense as a paradigm in distributed systems since communication between nodes is expensive and should be minimized. Hence, divide-and-conquer is usually not efficient. Instead, a more course-grained approach towards parallelism is taken by dividing large datasets on different computers (or nodes), letting each operate on its own chunk of data (to perform maps and generate partial results) and when its necessary reduce these partial results to a single node again.

6.2.1.4 Stencil

A stencil is a generalization of a map. Remember that we had two criteria for a map; it needs to be applied to each element of a collection independently and it needs to result in a collection of the same size. A stencil relaxes the first condition and allows the function to take more than one element of the collection as an input.

Stencils are often used in image processing in the form of filters. The canonical example is a median filter over an image, where the input and output is a two-dimensional array. The (i,j)'th entry of the output is the median value of the (i,j)'th entry and its surrounding values in the input matrix.

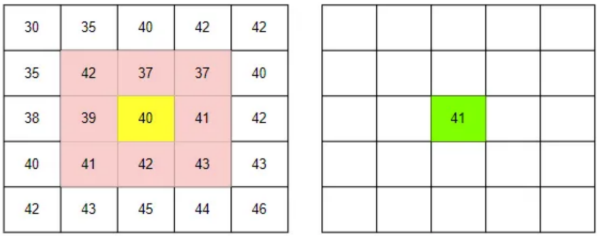


Fig. 32: In Green is the Output Value corresponding to the Yellow Input Value and its Surrounding: https://miro.medium.com/v2/resize:fit:1148/format:webp/1*-NVjiEYLFVOdRbxkihA8Qg.png

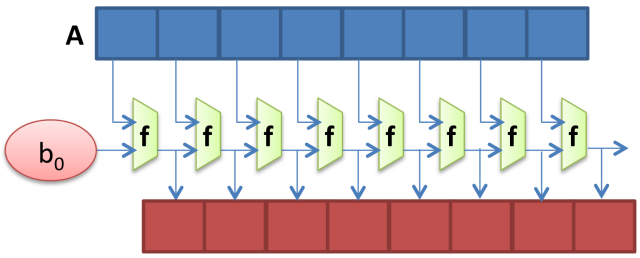
Stencils can be parallelized just as maps, since each output element can be computed independently. We need to make sure to not perform the computation in-place, else we might read elements as input that are actually already computed output. So, usually, the result needs to be a new collection.

6.2.2 Scan

A scan is a pattern, where:

- We are given a collection A and need to return a collection B.
- We are given a function f and the output B is computed according to the rule $B[i] = f(B[i-1], A[i])$.
- This rule is equivalent to the following: $B[i] = f(A[i], f(A[i-1], ..., f(A[0], b_0)))$. b_0 is some initial given value.

We can visualize this in the following way:



While the pattern is similar to a reduction, it seems inherently sequential. However, we will see that we can efficiently parallelize scan patterns where f is associative (because then we can implement a divide-and-conquer version) to get a $\mathcal{O}(log(n))$ span.

6.2.2.1 Scan Example: Parallel Prefix Sum

To get a better feeling for the pattern, we proceed with an example. Consider the prefix-sum problem, where we are given an input array and need to return an output array defined as:

$$output[i] = \sum_{k=0}^{i} input[k]$$

Here, we have $f(x, y) = x + y$ and $b_0 = 0$. A sequential Java implementation of the prefix-sum problem looks like this:

```
public int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i=1; i < input.length; i++)
        output[i] = output[i-1] + input[i];
    return output;
}
```

6.2.2.2 Parallelizing the Prefix Sum Algorithm

The span of this implementation is $\mathcal{O}(n)$, which allows for no parallelism. Let us try to improve upon it by using a divide-and-conquer approach. Assume we have an array of eight elements. Sequential prefix sum has a critical path containing seven additions. Imagine now we divide this array in two halves for which we compute their prefix-sum in parallel:

input	6	4	16	10	16	14	2	8
output	6	10	26	36	16	30	32	40

Both halves require three additions, which we can perform in parallel. In order to fix the result of the right half, we need to add 36 to each element. These four additions are independent though and can be

performed in parallel. Hence, the critical path contains only four additions. With this fix, we receive the correct result:

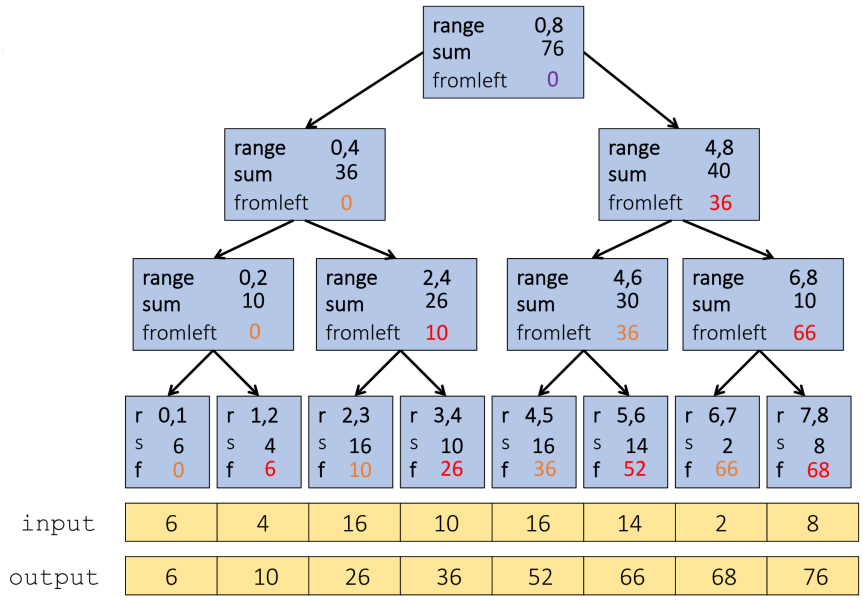
input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

Continuing the algorithm, let us try to write this as a classical divide-and-conquer algorithm, i.e. think about what the base case is and how to combine the partial results:

- The base case is the usual; a for loop sequentially computing the prefix sum for the assigned chunk. We already wrote the code for this above. This will compute the *local* prefix sum, in the sense that this will need to be corrected by adding the sum of all elements left of the assigned chunk. We need to think about this correction in the recursion.
- For the recursion, we need to fork and join a task for each half of the assigned chunk to compute the (local) prefix sum. But what do we need to do to combine the two results? We need to correct the right half somehow. The left half is okay, but to each element in the right half, we need to add the sum of all elements in the left half. Doing this sequentially means $\mathcal{O}(n)$ to combine results unfortunately, leading to an overall $\mathcal{O}(n)$ span. However, we want $\mathcal{O}(log(n))$. We cannot do this in a single pass.

We conclude that we have to do the computation in two passes. The plan is that the first pass computes the local prefix sums and the second pass adds the corrections. Two passes in both $\mathcal{O}(log(n))$ span means the overall span will still be in $\mathcal{O}(log(n))$. But now, how do we execute the second pass to correct the results? Let us think about *when* we need to add a correction:

When we initially split the array into two, we do not need to add a correction to the left chunk, since no array elements are to the left of it. But, to the elements of the right chunk, we will need to add the sum of all the elements of the left chunk. The beauty of recursion is that this happens again every time we further split the array. We always need to add the previous correction plus the sum of the elements of the chunk assigned to the left subtask. Let us draw the resulting computation tree on our previous example to understand what is going on:



The nodes of our computation tree are the tasks created. The `fromleft` field is the correction we need to apply. Remember that we said that the left subtask will not need to do additional correction, so a task can pass on its correction to the left subtask. To the right subtask however, it will need to add the sum of all elements in the range of the left subtask. When each node (or task) stores the sum of all the elements in its range, the correction can be done in the second pass using the following divide-and-conquer pattern:

```
if (end - start <= CUTOFF) {
    for (int i = start; i < end; i++) {
        resultArray[i] += correction;
    }
} else {
    int mid = (start + end) / 2;

    SecondPass left = new SecondPass(resultArray, correction, start, mid);
    SecondPass right = new SecondPass(resultArray, correction + sumOfLeftSubtask, mid, end);
    left.fork();
    right.fork();
    left.join();
    right.join();
}
```

The important things here are that we pass on the same correction to the left subtask and the correction plus `sumOfLeftSubtask` to the right subtask. This is more or less the final code for the second pass. However, in order to get `sumOfLeftSubtask`, we need to compute it in the first pass and store it somewhere for the second pass to find it.

So, let us now think about the first pass. We need to compute the local prefix sums, which we can simply do in the base case, nothing to do in the recursion. But we now also want to compute the sum of all elements in the range. This is the exact fork/join algorithm of summing up an array we previously implemented. Now we need to store this sum somewhere for each node in the computation tree. We can simply do this by *flattening* the tree into an array, just like with a heap. That is, we store the result of the first task (the root of the tree) at index 0. When a task stores its result at index i, its children will store it at indices $(2 * i) + 1$ and $(2 * i) + 2$ respectively. In the second pass, we can simply access these same indices again to retrieve the sums. Let us write the compute method of the first pass task:

```
class FirstPass extends RecursiveTask<Integer> {
    private final int CUTOFF = 1000;
    private int[] arr, res;
    private int start, end, id;

    FirstPass(int[] arr, int[] res, int start, int end, int id) {
        this.arr = arr;
        this.res = res;
        this.start = start;
        this.end = end;
        this.id = id;
    }

    protected Integer compute() {
        if (end - start <= CUTOFF) {
            int sum = arr[start];
            res[start] = arr[start];
            for (int i = start + 1; i < end; i++) {
                res[i] = res[i - 1] + arr[i]; // compute the (local) prefix sum
                sum += arr[i]; // compute the sum of the assigned elements
            }

            sums[id] = sum;
            return sum;
        } else {
            int mid = (start + end) / 2;

            FirstPass left = new FirstPass(arr, res, start, mid, 2 * id + 1);
            FirstPass right = new FirstPass(arr, res, mid, end, 2 * id + 2);
            left.fork();
            right.fork();
            int leftSum = left.join();
            int rightSum = right.join();
```

```
            sums[id] = leftSum + rightSum; // write the sum into an array
            return leftSum + rightSum;
        }
    }
}
```

The code is almost the same as the recursive array sum algorithm. The difference being that in the base case, we also compute the local prefix sum, we write the sum into a `sums` array and we also give the tasks an `id`. This is the final version of the first pass and we can now finally write the final version of the second pass:

```
class SecondPass extends RecursiveAction {
    private final int CUTOFF = 1000;
    private int[] res;
    private int start, end, correction, id;

    public SecondPass(int[] res, int correction, int start, int end, int id) {
        this.res = res;
        this.correction = correction;
        this.start = start;
        this.end = end;
        this.id = id;
    }

  protected void compute() {
    if (end - start <= CUTOFF) {
        for (int i = start; i < end; i++) {
            res[i] += correction;
        }
    } else {
        int mid = (start + end) / 2;

        SecondPass left = new SecondPass(res, correction, start, mid, 2 * id + 1);
        SecondPass right = new SecondPass(res, correction + sums[2 * id + 1], mid, end, 2 * id + 2);
        left.fork();
        right.fork();
        left.join();
        right.join();
    }
}
```

This is the final algorithm. Note that while we have three shared arrays (input array, result array and sum array), no two tasks write to the same array entry per pass and hence we do not need to introduce any synchronization. We can now write a method wrapping these two `ForkJoinTask` classes into a final algorithm:

```
public static int[] prefixParallel(int[] arr) {
    ForkJoinPool pool = new ForkJoinPool();
    int[] res = new int[arr.length];
    sums = new int[arr.length * 2];
    pool.invoke(new FirstPass(arr, res, 0, arr.length, 0));
    pool.invoke(new SecondPass(res, 0, 0, arr.length, 0));
    pool.shutdown();
    return res;
}
```

To perform two passes, we simply invoke the pool twice, once with the initial `FirstPass` task and then again with the initial `SecondPass` task. Note that at the time of the second invoke, all `FirstPass` tasks are finished, because they are all recursively joined. Also note that there are at most $2 * n$ tasks. The larger the sequential cutoff, the less tasks there are. But even with `CUTOFF=1`, there are only $2 * n - 1$ tasks, so it is sufficient to set the length of the `sums` array to $2 * n$, where n is the length of the input array `arr`.

To conclude, let us quickly go through the algorithm again:

- In the first pass, we compute both the local prefix sums, which we need to correct in the second pass, and also the sum of the array range each task gets assigned. We need these sums to compute the correction value in the second pass.
- In the second pass, the root gets an initial `correction` of 0. Then, the left subtasks gets the same `correction` and the right subtasks gets `correction` plus the sum of the range of the left subtask, which is stored at `sums[2*id + 1]`, because $(2 * id) + 1$ is the id of the left subtask. In the base case, this computed `correction` simply needs to be added to all elements in the assigned array range.

Note that the Java implementation of this algorithm is more involved than what is expected in the lecture. It is still a good exercise though.

6.2.2.3 Other Parallel Scans

We implemented a parallel scan algorithm using the example of the prefix-sum problem. But just like summing an array using fork/join was the simplest example of a parallel reduction, prefix-sum using fork/join is the simplest example of a parallel scan. Other examples of scans are:

- Minimum or maximum element to the left of i.
- Number of elements to the left of i satisfying some property.

Think about how we can parallelize these examples using the same template we used for parallel prefix-sum and what (minimal) changes we would have to make to the Java implementation.

6.2.3 Pack Pattern

The pack is a more complex pattern than the ones we have seen so far. However, the problem is simple to explain: We are given an array `arr` and need to return an array `res` containing all elements that fulfill some condition given by a boolean function f. We can visualize the pack pattern in the following way:

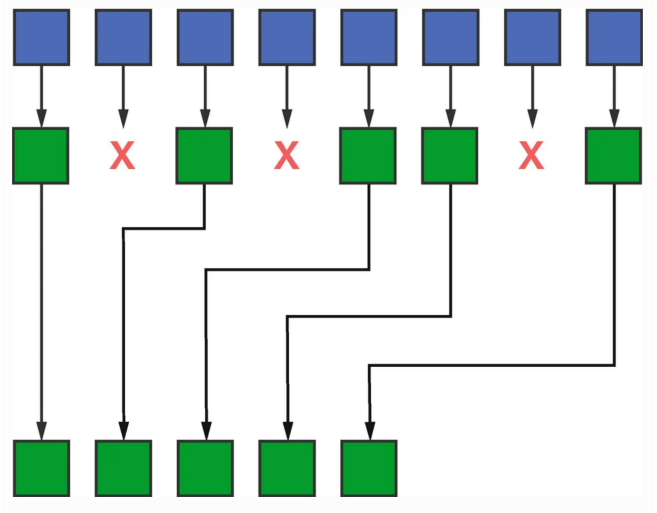


Fig. 33: https://link.springer.com/chapter/10.1007/978-1-4842-5574-2_14/figures/6

We can do the pack in three steps:

1. Compute a bit vector `bitVec`, where `bitVec[i]`=1 $\iff$ f(`arr[i]`)=1. That is, an entry of the bit vector is 1 if the corresponding element fulfills the condition and 0 otherwise. This is a simple map that can be computed in one pass using the fork/join framework with a $\mathcal{O}(log(n))$ span.
2. Now we need to find the position in the final array of each element. To do so, we compute a prefix sum `bitSum` on the bit vector. Then, `bitSum[i]` tells us how many elements in `arr[0],...,arr[i]` fulfill the condition. In the previous section we saw how we can parallelize prefix-sum to have an $\mathcal{O}(log(n))$ span as well.

3. Using the information of the second step, we can compute the place of an element in the output array in the following way:

```
if (bitVec[i] == 1) {
    res[bitSum[i]-1] = arr[i];
}
```

   This is again a map, which we can also implement with $\mathcal{O}(log(n))$ span.

Since all of the steps have span $\mathcal{O}(log(n))$, we can implement this parallel pack algorithm in $\mathcal{O}(log(n))$ span. Since 1. and 3. are simple maps and the code for 2. is given in the previous section, we omit the Java implementation here.

6.2.4 Case Study: Parallelizing QuickSort

Having learned to parallelize multiple patterns, we try to apply some of them to parallelize a familiar algorithm: QuickSort. First, we remember that sequential QuickSort is an in-place algorithm that runs in $\mathcal{O}(n * log(n))$ with the following steps:

1. Pick a pivot element: $\mathcal{O}(1)$
2. Partition the data (less than pivot, pivot, greater than pivot): $\mathcal{O}(n)$
3. Recursively sort left and right part: $2 * T\left(\frac{n}{2}\right)$.

We can trivially parallelize the recursive left and right sort using the fork/join framework. However, since partitioning the data is in $\mathcal{O}(n)$, we cannot get a span better than $\mathcal{O}(n)$. In order to improve this bound, we need to parallelize the data partitioning part.

To do so, let us think about what we really need to do to partition the array. All elements less than the pivot need to be selected and put to the *left* in the array, while all elements greater than the pivot need to be selected and put to the *right* in the array. However, if we allow this to not be performed in-place, this can be achieved with two packs: We pack all elements less than the pivot into a new array and all elements greater than the pivot into another new array. Then, we can combine those back into the original array and put the pivot element between them. Thanks to our parallel pack algorithm, this means $\mathcal{O}(log(n))$ span to partition the array with $\mathcal{O}(n)$ additional memory because the packs are not in-place.

Our recursion equation becomes $T(n) = \mathcal{O}(log(n)) + T\left(\frac{n}{2}\right) = \mathcal{O}((log(n))^2$, a great asymptotic improvement over the sequential $\mathcal{O}(n * log(n))$.

We see that a repertoire of some basic parallel patterns helps us effectively parallelize more complex algorithms. Implementing this parallel QuickSort in Java is more involved (remember that we had to code around 80 lines for parallel prefix-sum, which is just a part of the parallel pack, which in turn is just a part of parallel QuickSort) and it is enough for us to realize how it could be done, since parallelizing algorithms in Java is not the main focus of the course.

6.2.5 Parallelizing Algorithms on Different Datastructures

So far, we always assumed arrays as our datastructure. What is so special about arrays though? It is their property of being able to access each element in $\mathcal{O}(1)$.

6.2.5.1 Span of Divide-And-Conquer Algorithms

Let us recall what made parallel divide-and-conquer so effective. It was that we could reduce the span from $\mathcal{O}(n)$ to $\mathcal{O}\left(log(n)\right)$, which is the *depth* of the task graph. Consider a critical path in this divide-and-conquer task graph (remember it looks like a binary tree). We make three observations:

- There are $\mathcal{O}\left(log(n)\right)$ vertices on the path (this is the depth of the task graph).
- The last vertex on the path is the computation of the base case. Let us say this takes $\mathcal{O}(X)$ as it depends on the operation we want to compute.
- All other vertices on the path correspond to dividing up the data on subtasks and then combining the results. Time required to divide the work depends on the datastructure, so let us say it takes $\mathcal{O}(Y(n))$. Let us also say that combining results takes $\mathcal{O}(Z(n))$.

The time it takes to compute this critical path is thus $\mathcal{O}\left(log(n) * (Y(n) + Z(n)) \ + \ X\right)$, because we have to divide up the data $log(n)$ times in $\mathcal{O}(Y(n))$ each, compute the base case once in $\mathcal{O}(X)$ and recombine results in $\mathcal{O}(Z(n))$ $log(n)$ times. The time it takes to compute the critical path is by definition the span, so this is also the span of a divide-and-conquer algorithm. Note however that the O-notation means it is an upper bound. In this case, the bound is not tight and depending on the functions X, Y and Z, we can get a better bound than the one suggested above.

Since for arrays, dividing up the work is possible in $\mathcal{O}(1)$, that is, $Y \equiv 1$, the span of divide-and-conquer algorithms (on arrays) is $\mathcal{O}(log(n))$ for all operations, for which the base case (X) and recombining results (Z) is in $\mathcal{O}(1)$. That is the case in particular for reductions and maps.

6.2.5.2 Linked Lists

For linked lists, accessing an element takes $\mathcal{O}(n)$. Imagine we would want to sum all elements of a linked list using a divide-and-conquer approach. In order to divide the list in half, a subtask would have to traverse to the second half of the assigned chunk, which takes $\mathcal{O}(n)$. Hence, $Y \equiv n$ and thus we cannot get a span less than $\mathcal{O}(n * log(n))$ using divide-and-conquer on linked lists. Divide-and-conquer does not make sense on a linked list due to the linear traversal time and we would be better of simply using a naive approach, where can at least get to $\mathcal{O}(n)$.

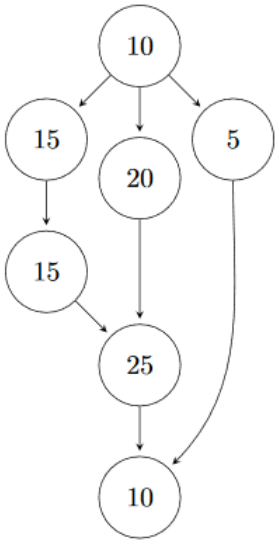
Not all is bad though. Consider a long computation that takes $\mathcal{O}(X)$, which we want to compute for each element of the linked list. Sequentially, this takes $\mathcal{O}(n * X)$. The parallel span is $\mathcal{O}(n + X)$ though. The critical path here is the task that needs to traverse to the last element of the list ($\mathcal{O}(n)$) and then compute the function ($\mathcal{O}(X)$).

6.2.5.3 Balanced Trees

Divide-and-conquer parallelism works well in balanced trees. Imagine we want to sum all elements in the tree. Dividing up the work can be done by forking the left and right child to process in parallel. Dividing up the work hence takes $\mathcal{O}(1)$ and we get $\mathcal{O}(h)$ span, where h is the height of the tree. In balanced trees, this is logarithmic in n.

6.3 Exercises

3. Task Graph. The following figure shows the task graph for an algorithm. The number in each node denotes the execution time per task. What is the maximum overall achievable speedup that can be achieved by parallelism when the algorithm runs once compared to sequential execution? How many processors are required to achieve this speedup?



4. Fork/Join Framework. Given an array of integers, your task is to find the maximum subarray sum among all possible subarrays. For example,

$$\{2, -4, \mathbf{1, 9, -6, 7}, -3\} \rightarrow 11 \text{ (marked in } \mathbf{bold}\text{)}$$

Extend the following class such that it computes the maximum subarray sum as described above.

```
public abstract class MaxSubArraySum extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1L;
    int start;
    int length;
    int[] input;

    public MaxSubArraySum(int start, int length, int[] input){
        this.start = start;
        this.length = length;
        this.input = input;
    }

    public abstract Integer compute();
}
```

6.4 Solutions

3. Task Graph. The formula for speedup is $S_P = \frac{T_1}{T_P}$. We know that T_1 is the simply the sum of the execution times of all nodes, i.e. $T_1 = 100$. We also know that the best-possible execution time achievable through parallelization is limited by the length of the critical path. The critical path has length 75, i.e. $T_\infty = 75$. Inserting these values into the formula for speedup gives us:

$$S_\infty = \frac{T_1}{T_\infty} = \frac{100}{75} \approx 1.33$$

We find that two processors are enough to achieve this speedup.

4. Fork/Join Framework. We stick to the generic recipe for the Fork/Join framework. The difficulty with this exercise is the merging of the results of the two subtasks, as the maximum subarray sum might contain elements of both array partitions. We therefore need to additionally compute the maximum subarray sum which crosses the middle border and return the maximum of the three sums.

```java
public class MaxSubArraySumForkJoin extends MaxSubArraySum{

    private static final long serialVersionUID = 1L;

    public MaxSubArraySumForkJoin(int start, int length, int[] input, int CUTOFF) {
        super(start, length, input, CUTOFF);
    }

    public Integer compute(){
        // Check base case
        if(length <= CUTOFF){
            return MaxSumHelper.MaximumSum(input, start, start+length-1);
        }
        // Split work
        int mid = length / 2;
        MaxSubArraySumForkJoin l = new MaxSubArraySumForkJoin(start, mid, input, CUTOFF);
        MaxSubArraySumForkJoin r = new MaxSubArraySumForkJoin(start+mid, length-mid, input, CUTOFF);
        // Compute maximum subarray sums of left and right partitions
        l.fork();
        int rightMax = r.compute();
        // Find maximum subarray sum crossing middle border
        int rightMidMax = 0, sum = 0;
        for(int i = start+mid; i<start+length; i++){
            sum += input[i];
            if(sum > rightMidMax)
                rightMidMax = sum;
        }
        int leftMidMax = 0;
        sum = 0;
        for(int i = start+mid-1; i>=start; i--){
            sum += input[i];
            if(sum > leftMidMax)
                leftMidMax = sum;
        }
        // Combine results
        return Math.max(leftMidMax + rightMidMax, Math.max(l.join(), rightMax));
    }
}
```

7 Locking

In this chapter, we briefly return to the topic of critical section programming. Remember that a critical section is a part of a (multi-threaded) program, which only one thread should execute at a time.

In the previous chapter, our goal was to parallelize algorithms, but we never needed to worry about shared memory and critical sections. This is because of the special structure of divide-and-conquer programs: Although the datastructure was shared, each task had a chunk of the datastructure only it (and its subtasks) accessed. Whenever a subtask was `fork()`ed, it was also `join()`ed again and the memory was accessed only after the join completed. Hence, there were never concurrent accesses to a shared memory location. However, there are plenty of programs where memory is shared and concurrent writes to the same data are allowed. So, after focusing on performance (decreasing span) in the previous chapter, we focus on shared memory concurrency again, where correctness is the focus.

7.1 Approaches Towards Sharing Resources

There are three general approaches to take when dealing with shared data across threads:

1. **Immutability**: The datastructure is simply not allowed to be changed (mutated). That is, no writes are allowed. Concurrent reads are not a problem, so no synchronization is required. We should make our datastructures immutable wherever possible.
2. **Isolated Mutability**: The data is allowed to be changed, but no two threads can access the same memory location. Due to this constraint, we do not need to introduce synchronization and cannot have conflicting accesses. This approach is used in divide-and-conquer, where each thread/task only accesses its assigned part of the datastructure.
3. **Mutable/Shared Data**: The data is freely allowed to be changed with no restrictions. Now, multiple threads can access the same memory locations concurrently and we need to introduce synchronization, which we can do for example by using locks. We now focus on this scenario in this chapter.

So, for the rest of the chapter, we assume mutable shared data for which we need to synchronize accesses.

7.2 Locks

When we have shared mutable data, accesses to this data are a critical section. We must ensure that only one thread accesses the data at a time, a property we already introduced as *mutual exclusion*. The mutual exclusion property can be provided by locks in Java. Until now, we always used the Java synchronized keyword, which provides mutual exclusion to a code block by using the *intrinsic lock* or *monitor lock* of objects. Remember that the intrinsic locks allowed us to simply wrap a critical section into a `synchronized()` block:

```
synchronized(o) { // o is some Object
    // critical section code
}
```

To truly understand how the `synchronized` keyword relates to locks, let us write pseudo-code of what actually happens here:

```
Lock l = o.getIntrinsicLock(); // pseudo-code, this method doesn't exist
l.lock();
/* critical section code
*/
l.unlock();
```

The executing thread obtains the intrinsic lock of the `Object o` and acquires it. After the critical section is over, the thread releases it again.

We realize that the `synchronized` keyword provides an abstraction on top of locks. However, we now want the flexibility of (external) locks and go beyond this abstraction. Using external locks, we can get the same semantics with the following code:

```
Lock l = new Lock();
l.lock();
/* critical section code
*/
l.unlock();
```

This is now valid Java code, albeit not yet entirely correct (we will soon see why).

While using external locks removes the abstraction layer, it is arguably more intuitive than simply wrapping the critical section into a `synchronized` block, since it is very descriptive; we acquire some object (the `Lock`) that no one else can acquire while we hold it, until we release it again. Think of the `Lock` like a talking stick used in discussions with a lot of people. Only the person holding the stick is allowed to talk and no other person is allowed to speak until he or she acquires the stick. So, when we hold this object, we are guaranteed to be the only one holding it and can thus execute critical section code, like writing to shared data. We will learn later in the course how such locks are actually implemented to guarantee this. At this point, we can simply take locks as a given primitive with the semantics of guaranteeing mutual exclusion.

7.2.1 Using External Locks vs Intrinsic Locks

Before we can appreciate the benefits of using external locks over the intrinsic ones (intrinsic ones are used by the `synchronized` keyword), let us introduce an example: As systems engineers, we have to design a banking system for the struggling JVM (Java Virtual Money Bank). To start, we write a simple `transferMoney(Account from, Account to, int amount)` method:

```
public class BankingSystem {
    public boolean transferMoney(Account from, Account to, int amount) {
        if (from.getBalance() < amount || to.getBalance() + amount < 0) {
            return false;
        } else {
            from.setBalance(from.getBalance() - amount);
            to.setBalance(to.getBalance() + amount);
        }
        return true;
    }
}

interface Account {
    public int getBalance();
    public void setBalance();
}
```

The `transferMoney` method checks whether the charged account has enough balance and if it does, sets the balance of the two accounts accordingly. If we use this code with multiple threads and transfer money concurrently, all sorts of bad interleavings can occur. Try to find some possibilites of bad interleaving that would be allowed without any synchronization.

With our previous knowledge, the approach to fix this is to simply decorate the method with the `synchronized` keyword:

```
public synchronized boolean transferMoney(Account from, Account to, int amount) {
    if (from.getBalance() < amount || to.getBalance() + amount < 0) {
        return false;
    } else {
        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);
    }
    return true;
}
```

This approach prevents any bad interleavings. If you found some bad interleavings that were possible before, try to reconstruct why they are not possible anymore.

A possible scenario without synchronization is for example that two threads transfer from some `Account a` with balance 10 to `Account b` concurrently. That is, both threads execute the function call `transferMoney(a, b, 10)` concurrently. A possible interleaving is that both threads find that `a` has sufficient balance and then both decrease the balance by 10, leading to negative balance for `a`, which should not be allowed.

How would we now create the same effect using external locks? Try to write down the code yourself. The approach is quite straightforward: We create a `Lock` object as a field of the `BankingSystem` class and acquire it at the beginning of the `transferMoney` method:

```
public class BankingSystem {
    private Lock lk = new Lock();

    public boolean transferMoney(Account from, Account to, int amount) {
        lk.lock();
        try {
          if (from.getBalance() < amount || to.getBalance() + amount < 0) {
              return false;
          } else {
              from.setBalance(from.getBalance() - amount);
              to.setBalance(to.getBalance() + amount);
          }
        } finally {
            lk.unlock();
        }
        return true;
    }
}
```

Notice the try-finally-block. The purpose of this is that the finally-block is executed in any case, either when the try-block finished or when an exception occurred in the try-block. If we do not put the `lk.unlock()` in a finally-block, the thread causing an exception does not release the lock and thus blocks all threads waiting for it. Unlocking the lock again is critical. Using `synchronized`, we never had to think about this, because the monitor lock was released automatically at the end of the `synchronized` block. Apart from the try-finally-block, the modifications in this snippet should not be a big surprise.

7.2.2 Are Java `Locks` reentrant?

Locks provide mutual exclusion, meaning only ever one thread can hold it at any point in time. Remember that for intrinsic locks, we mentioned that they are reentrant. This means that a thread holding a monitor lock can acquire it multiple times (for example with nested `synchronized` blocks or by calling another `synchronized` method from within a `synchronized` block). When we simply instantiate a `Lock` object in Java, this is not going to be a reentrant lock:

```
Lock lk = new Lock(); // this lock is not reentrant
```

However, the Java lock library provides a `ReentrantLock` class that has the reentrant property and can else be used in the same way we have used the `Lock` class before.

```
ReentrantLock lk = new ReentrantLock(); // this lock is reentrant
```

When a thread acquires a `ReentrantLock` n times, it also needs to release it n times again to make it available for other threads again. Internally, this is implemented with a counter; when a thread acquires a `ReentrantLock` it already obtained, the lock incremenents an internal counter. Each `unlock()` decrements the counter. The lock is only available for other threads when the counter reached 0.

```
ReentrantLock lk = new ReentrantLock(); // this lock is reentrant
lk.lock(); // obtain the lock
lk.lock(); // increment the internal counter to 2
lk.unlock(); // decrement internal counter to 1. Lock unavailable for other threads
lk.unlock(); // decrement counter to 0. Lock now available for other threads
```

7.2.3 Concluding `Locks` vs. `Synchronized`

Let us list some of the usability differences between the `Lock` API and the `synchronized` keyword in a table:

	`Synchronized`	`Lock` API
Acquire	Automatic (when block begins)	Manually (call to `lk.lock()`)
Release	Automatic (after block ends)	Manually (call to `lk.unlock()`)
Scope	Limited by synchronized block.	From `lk.lock()` to `lk.unlock()` (can range across method calls).
Reentrant?	Yes	Only `ReentrantLock` class.

Arguably the key difference between the two is the flexibility. Being able to `lock()` and `unlock()` from anywhere allows us for example to obtain a lock in a method and release in a different one. It also allows us to release the locks in an order different than the reverse acquire order:

```
Lock lk = new Lock();
Lock lk2 = new Lock();
lk.lock();
lk2.lock();
lk.unlock();
lk2.unlock();
```

With `synchronized`, this is impossible. Whenever we have nested synchronized blocks, the monitor locks are acquired in some order and released in reverse order:

```
sychronized(o) { // obtain monitor lock of o
    sychronized(o2) { // obtain monitor lock of o2
        /* some code */
    } // release monitor lock of o2
} // release monitor lock of o
```

Later in the course we will make use of this. At this point, we just note that external locks are more flexible than using the `synchronized` keyword.

7.3 Locking Granularity

Granularity refers to how much functionality a lock guards. In our banking system, we use the same lock for the entire system. That is, no matter between what accounts a transfer happens, the thread executing this transfer would simply acquire a *global* lock. Such an approach is called *coarse-grained* locking, because the lock guards the entire banking system functionality. When a thread wants to do a transfer, it acquires this lock and hence all other threads wanting to execute any transfer will have to wait for this thread to finish and release the lock again.

We could also have a different lock for each account object. Then, a thread would acquire the locks of the two accounts it wants to make a transfer between. Here, the locks guard *less* functionality, since other threads wanting to transfer money between two different accounts could still do so in parallel, because these other accounts are guarded by their own locks. This approach is called *fine-grained* locking.

7.3.1 `BankingSystem` with Fine-Grained Locking

When we want to implement this fine-grained locking in our `BankingSystem` example, we need to modify the `Account` class. Specifically, we need to:

- Define a `Lock` class attribute to have a lock for each `Account` instance..
- Provide a `getLock()` method that returns the `Lock` of the `Account` instance.

Let us write down the code for this:

```
public class Account {
    private Lock lk = new Lock();

    public Lock getLock() {
        return this.lk;
    }
    ...
}
```

Now we can rewrite the `transferMoney` method in the `BankingSystem` class:

```
public boolean transferMoney(Account from, Account to, int amount) {
    from.getLock().lock();
    to.getLock().lock();
    try {
        if (from.getBalance() < amount || to.getBalance() + amount < 0) {
            return false;
        } else {
            from.setBalance(from.getBalance() - amount);
```

```
            to.setBalance(to.getBalance() + amount);
        }
    } finally {
        from.getLock().unlock();
        to.getLock().unlock();
    }
    return true;
}
```

This looks simple enough. Basically, we just have to acquire and release two locks (one per account involved) instead of one. With this approach, we can get potentially get much more parallelism if we have a large banking system with a lot of concurrent transfers happening and we had to add very few additional code. This code is not entirely correct however. There is a possibility of a bad interleaving for certain concurrent transfers. Try to find it.

7.3.2 Hidden Bugs with Fine-Grained Locking

The problem with above code is that a deadlock can occur if we have for instance the following concurrent function calls:

```
transferMoney(a, b, 100); // imagine this runs in thread t1
transferMoney(b, a, 100); // concurrently, this runs in thread t2
```

If you did not yet find what can go wrong, consider again how a deadlock can happen with these two concurrent function calls.
Consider the following interleaving:

`t1`:

```
a.getLock().lock(); // 1.

b.getLock().lock(); // 3.
```

`t2`:

```
b.getLock().lock(); // 2.

a.getLock().lock(); // 4.
```

Note that for `t1`, we have `from == a` and `to == b` and for `t2` the other way around, which is the problem here. The following is happening:

1. `t1` acquires the lock of `Account a`.
2. textttt2 acquires the lock of `Account b`.
3. `t1` wants to acquire the lock of `Account b`, which `t2` already holds. `t1` is blocked until `t2` releases the lock on `b`.
4. `t2` wants to acquire the lock of `Account a`, which `t1` already holds. We have reached a deadlock, since both threads wait for the other to release its lock, which results in a circular dependence with no way out.

This requires a very specific interleaving that will in practice occur extremely rarely, which means the bug is hard to find when testing, which is the danger of fine-grained locking.

Think about how this can be circumvented. The first thought might be to introduce some deadlock detection, such that one thread releases its lock again when such a scenario occurs. However, such a solution is hard to test, results in complex code and is prone to errors itself. A simple solution to prevent this is to introduce a global ordering on the accounts (for example by account number). Then, we would always lock the account with the smaller account number first. Now, a circular dependence is not possible anymore (to see this, try to find concurrent function calls that enforce a deadlock. You will see that it is not possible to get circular dependencies).
In Java, we can implement such an ordering by giving IDs to the accounts:

```
class Account {
    private int id;
    private int balance;

    public int getId() {
        return this.id;
    }
    ...
}
```

Now we can make use of this in the `transferMoney` method:

```
public  boolean transferMoney(Account from, Account to, int amount) {
    Account first, second;

    // account with the smaller id will be locked first
    if (to.getId() > from.getId()) {
        first = from;
        second = to;
    } else {
        first = to;
        second = from;
    }

    // rest of code is the same
    first.getLock().lock();
    second.getLock().lock();
    try {
        if (from.getBalance() < amount || to.getBalance() + amount < 0) {
            return false;
        } else {
            from.setBalance(from.getBalance() - amount);
            to.setBalance(to.getBalance() + amount);
        }
    } finally {
        first.getLock().unlock();
        second.getLock().unlock();
    }
    return true;
 }
```

We simply fix the order of the accounts (always first lock the one with the smaller account number) and then execute the same code as before. The previously mentioned deadlock problem is now solved.

7.3.3 Concluding Locking Granularity

We can see that fine-grained locking allows for more parallelism, but it also requires more work to ensure correct behaviour. In general, it is wise to start with coarse-grained locking and to try improving parallelism by moving to more fine-grained locking later, when *contention* becomes an issue. In our banking system example, fine-grained locking only provides an advantage if many transfers happen in parallel and hence threads have to wait often. If there is no contention and concurrent transfers are rare, there is not much to be gained by moving to fine-grained locking. On the contrary, it is prone to introducing hard-to-detect bugs, like the possible deadlock we previously saw.